UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

# GENETIC ALGORITHMS APPLIED TO VIDEOGAME TESTING

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

TOMÁS ANDRÉS AHUMADA BALART

PROFESOR GUÍA:
ALEXANDRE BERGEL

MIEMBROS DE LA COMISIÓN:
JEREMY BARBAY LEFEVRE
LUIS MATEU BRULÉ

SANTIAGO DE CHILE
2020

# RESUMEN

En los últimos años, los métodos de testeo usados en videojuegos han consistido en expandir técnicas clásicas de testeo de software, como Unit Testing por ejemplo. Sin embargo en la época moderna, la mayor parte del testeo de un juego es llevada a cabo por un grupo de testers humanos que interactúan con el software y hardware.

Hay casos en que un tester logra reproducir un bug una vez pero por diversos motivos no es capaz de reproducirlo después. Esto puede ser porque el bug en particular puede ser difícil de reproducir para un ser humano (puede requerir inputs muy precisos), o la información obtenida al reproducirlo la primera vez es muy escasa. Volver a producir este tipo de errores para un tester humano puede tomar mucho tiempo.

En esta memoria se propone una técnica basada en algoritmos genéticos, una técnica perteneciente al reino de la Inteligencia Computacional, para resolver este tipo de problemas. Usaremos algoritmos genéticos para crear secuencias de inputs a correr sobre un juego, y obtener feedback del juego para computar la función de fitness. Al definir una función de fitness con datos de un juego, de modo que su valor aumente mientras más cerca esté la secuencia de inputs de reproducir el error; podemos producir una secuencia de inputs que reproduce el bug con algoritmos genéticos.

Desarrollamos un grupo de codificaciones para representar secuencias de inputs junto a sus respectivas funciones de crossover y mutación. Estas codificaciones fueron probadas y comparadas en tiempos de reproducción y total de fitness obtenida al realizar una tarea simple sobre una implementación del juego Asteroids en Github. Esto llevó al desarrollo de dos codificaciones, que fueron utilizadas para reproducir bugs en dos juegos open-source.

Para cada juego logramos reproducir al menos uno de los bugs que nos enfrentamos, y analizamos los resultados para obtener más información sobre ellos. Contactamos a los desarrolladores y, guiados por ellos, realizamos un análisis más profundo de los resultados para obtener más detalles sobre los bugs usando nuestro software. La información obtenida de los análisis ayudó a los desarrolladores a confirmar teorías que tenían sobre estos problemas y a parchar estos bugs.

El software desarrollado en la memoria es un primer paso en la aplicación de algoritmos genéticos de esta forma en la evolución de secuencias de inputs. A través de los experimentos llevados a cabo, algunas guías fueron desarrolladas para usar esta técnica para reproducir errores. Para que las herramientas desarrolladas sean usadas en la práctica, algunas funcionalidades tienen que ser añadidas para que los desarrolladores de juegos puedan visualizar y editar los contenidos de una secuencia de inputs.

# ABSTRACT

Over recent years, automated testing in videogames has consisted of expanding classical software techniques like Unit Testing to these Real-Time Graphic Applications. However, in the modern era most of the testing related work is done by a group of human testers that interact with the software and hardware.

There are times that a tester manages to produce a bug, but then fails to reproduce it for various reasons. The bug could be hard to reproduce for a human (a frame-perfect glitch for example), or the known information about it is very limited. Reproducing these bugs in order to understand them can take a lot of time for a human tester.

This thesis proposes a technique based on genetic algorithms, a technique that belongs to the area of Artificial Intelligence, to solve these problems. We use genetic algorithms to produce sequences of inputs to run over a game, and get feedback from the game in order to compute the fitness function. By designing a fitness function using game data so its value increases the closer an input sequence is to reproduce a glitch; we can produce an input sequence that reproduces the glitch with the genetic algorithm.

We also propose a group of encodings to represent input sequences along with their respective crossover and mutation methods. These encodings were tested and compared in reproduction time and fitness achieved when performing a simple task over an Asteroids implementation found on Github. This led to the development of two definitive encodings, which were then used to reproduce reported issues in two other open-source games.

For both of these games, we managed to reproduce at least one of the issues tackled, and then we analyzed the results to get information about them. We contacted the developers, and performed further analysis guided by them in order to obtain a better understanding of the issues using the software developed. Our feedback helped the developers confirm theories they had and patch these issues.

The software created is a first step in applying genetic algorithms in this manner to the evolution of input sequences. Through experimentation, some guidelines were developed to use this technique to reproduce issues. In order for the tools created to be used in practice, some features that help developers visualize and edit the content of an input sequence are required.

# Contents


**1 Introduction** **1**
- 1.1 Testing in Video games . . . 1
- 1.2 Motivation . . . 2
- 1.3 Description of the Solution . . . 3
- 1.4 Results . . . 3

**2 Theoretical Framework** **5**
- 2.1 Genetic Algorithms . . . 5
  - 2.1.1 Genes and Individuals . . . 5
  - 2.1.2 Fitness Functions . . . 6
  - 2.1.3 Selection . . . 6
  - 2.1.4 Crossover . . . 7
  - 2.1.5 Mutation . . . 7
  - 2.1.6 Elitism . . . 8
  - 2.1.7 Multi-Objective Fitness Functions . . . 8
- 2.2 Technologies Used . . . 10
  - 2.2.1 Python . . . 10
  - 2.2.2 Lua . . . 10
  - 2.2.3 LÖVE . . . 10

**3 Problem** **11**
- 3.1 Hard-to-Reproduce Issues . . . 11
- 3.2 Solution Requirements . . . 12
  - 3.2.1 Basic Requirements . . . 12
  - 3.2.2 Quality Requirements . . . 13
  - 3.2.3 Desirable Features . . . 13

**4 Solution** **14**
- 4.1 LÖVE Wrapper . . . 14
  - 4.1.1 The Wrapper's Pipeline . . . 14
  - 4.1.2 Input Sequence Encoding . . . 16
  - 4.1.3 Fitness Functions . . . 17
- 4.2 Genetic Algorithm Software . . . 18
  - 4.2.1 Genes . . . 18
  - 4.2.2 Individuals . . . 19
  - 4.2.3 Testers . . . 20

4.2.4 Timeline . . . . . 21

**5 Validation** **23**
5.1 LuaAsteroids . . . . . 23
5.1.1 Results . . . . . 24
5.2 Journey to the Center of Hawkthorn . . . . . 26
5.2.1 Moving Platform Experiments . . . . . 26
5.2.2 Stay Experiment . . . . . 29
5.2.3 Out of Bounds Experiment . . . . . 32
5.2.4 Experiment Conclusions . . . . . 37
5.3 Zabuyaki . . . . . 37
5.3.1 D545 Task Point 1 . . . . . 39
5.3.2 D545 Task Point 2 . . . . . 43
5.3.3 D694 Task . . . . . 47
5.4 Time Analysis . . . . . 54
5.4.1 Testing Time Analysis . . . . . 55
5.4.2 Reproduction Time Analysis . . . . . 56
5.4.3 Time Acceleration Feature Analysis . . . . . 58
5.4.4 Experiment Conclusions . . . . . 59

**6 Conclusion** **61**
6.1 Summary of the Work Done . . . . . 61
6.2 Objective Completion Review . . . . . 62
6.3 Results Analysis . . . . . 63
6.4 Discussion . . . . . 64
6.5 Future Work . . . . . 65

# Chapter 1

# Introduction

## 1.1 Testing in Video games

Video games are graphic software applications made mainly for entertainment purposes. They are usually encapsulated in genres that share some characteristics like camera configuration, gameplay structure, themes, etc. However in spite of two games being in the same genre, these games can play, feel and look completely different from one another, as there is no defined formula on how games are made.

Game Development teams are divided in sub-teams, each specialized in a specific discipline. In order to create videogames, a very diverse set of skills is required, apart from programming. Therefore in a regular team it is expected to see a management team, a marketing team, an art team, a music and audio team, a game design team, a programming team and a Quality Assurance (or QA) team.

QA teams in video game development studios are usually formed by programmers or computer science engineers and a large number of testers. Testers are in charge of playing different levels of a game in order to find bugs or glitches, give feedback of its playability and other related tasks. The QA team on the other hand is in charge of guiding the testers on what parts of the game to test and what behaviors to try.

A video game tester's job includes: checking the game's reaction to plugging and unplugging the controls or other hardware, navigating menus and playing the game in search of errors. Some of the tester's responsibilities include tedious or repetitive tasks [4], for example playing a level in a game multiple times, in order to detect errors or calibrate variables.

For this purpose, automated testing in video game studios is becoming a common complementary method to manual testing during development. Concepts from classic automated software testing are being extended to be used in these applications. Unit Testing and Test-driven Development are two examples of these concepts.

The use of automated testing has helped many development studios to: reduce the amount of critical bugs encountered during development; give space to the testing team to test more

complicated behaviors; and reduce the amount of glitches found in the final version of the game. For example, according to Jim Merril in his article "Automated Testing for League of Legends" [15], the team behind the game League of Legends uses these techniques and runs approximately 100,000 test cases a day. It is shown that a large number of bugs are found this way, and get resolved around 8 times faster compared to when they are encountered manually.

There is another tool whose use is not standard in the whole industry, but has proven to help greatly during development. It is called Instant Replays; a feature that works with deterministic games and consists of recording the inputs of a session of playing a game, and saving it in a file to be reproduced later from within the game. Its use was documented in a 2001 article in Gamasutra [6] and was recently covered in Brian Provinciano's talk "Automated Testing and Instant Replays in Retro City Rampage" in GDC 2015 [17].

Nowadays there is a small amount of tools that apply automated testing in video games (implemented in video game engines) which have achieved good results. Therefore, most studies in video game testing are centered on creating a culture in the programming team to test features before adding them to the game's current build. Even though these practices boost the quality of the finished product greatly, the development of new testing tools would also greatly help in this regard.

## 1.2 Motivation

Genetic Algorithms (or GA for short) are optimization algorithms based on the concepts of Evolution and Natural Selection [11]. These algorithms are used to find local optimum solutions for multi-variable functions, along with the values that generate them. The only requirement for their use is a multi-variable function which domain consists of a space with finite dimension.

Given the characteristics of these algorithms, they can be used to generate input sequences that optimize a function, which depends on in-game variables. By having access to the internal variables of a game, a fitness function can be developed so input sequences created by the GA perform a defined target behavior detectable by the game. This target behavior can be designed to reproduce a bug or to perform a simple task within the game.

Performance data, physics engine data and game logic are some of the sources of variables that can be used. Performance variables can include Frames per Second, CPU and RAM usage. Physics Engine variables, like speed and acceleration, are interesting because these are subjected to approximation errors; so taking them to certain values can create unexpected behaviors that causes bugs to occur.

We explored the possibilities of applying Genetic Algorithms to automatic testing in videogames. We then used the results of our research to reproduce bugs in some open-source games. The tools were updated between the testing of different games to facilitate their use and give them more functionality when necessary.

The objective of this research is to create a tool for game developers to test their game

using these types of algorithms. Using these techniques, they can save time in the testing phase of a game, by using the tool to do the more repetitive and simpler tasks while the human testers focus on more complex work. For instance, automatizing a repetitive task such as attacking an enemy with every weapon, or in every scenario of the game will save a lot of human time.

## 1.3 Description of the Solution

In order to achieve the objectives enunciated, we developed a wrapper for the LÖVE framework in Lua (the LÖVE Wrapper), in order to run predefined input sequences. The wrapper is able to compute user defined functions given the game's internal parameters. On top of this, the wrapper can compute additional metrics for a future comparison with the fitness function.

We also implemented a program that uses genetic algorithms to generate input sequences as GA individuals. This program uses the wrapper to calculate the fitness of each one of these sequences. It also saves the input sequence that produced the best results.

We created several different encodings for representing input sequences as individuals; which were then polished in order to create one that worked well with Genetic Algorithms. The encodings varied in complexity, structure, and expressiveness. Through experimentation, we developed some encodings that are able to evolve with the GA.

We also implemented some general performance fitness functions for the LÖVE Wrapper. These performance functions are not game-dependant and work on any game compatible with the wrapper. These functions include: memory usage functions, input counting functions and time elapsed functions.

## 1.4 Results

After the development of the software mentioned, we used it to test a total of 3 games. These games were: luaAsteroids[1], Journey to the Center of Hawkthorn[2] and Zabuyaki[3]. We tested the first game in order to compare possible encodings for representing input sequences. The second and third games were tested to reproduce bugs found by the developers, in order to obtain more information about them and to report it to the developers. Additionally, we performed some experiments to measure the time that the execution of the software takes on these last two games.

After developing the encodings, through experimentation we determined that two encodings in particular were the ones that produced the best results when reproducing issues. The end of this document features an alternative use for the unused encodings.

At least one issue was reproduced for every game tested. Various fitness functions were

[1] `https://github.com/Lang-CS/luaAsteroids`
[2] `https://projecthawkthorne.com/`
[3] `https://www.zabuyaki.com/`

designed for the reproduction of these issues. Based on the results of the experiments, we defined some guidelines for designing fitness functions to solve issues using the individuals developed. Finally, the advantages and disadvantages of using these tools are presented compared to having human testers test everything.

# Chapter 2

# Theoretical Framework

## 2.1 Genetic Algorithms

Genetic Algorithms is a technique that has evolved through many decades. One of the first studies focused on applying Darwin's Natural Selection and Evolution theory was Alex Fraser in "Simulation of Genetic Systems by Automatic Digital Computers" (1957) [8]. In this text he explains how to apply the Monte Carlo method to Genetic Problems using the Digital Computer SILLIAC.

These algorithms started to become more popular thanks to the work of John Holland with his book "Adaptation in Natural and Artificial Systems" (1975) [11]. Where he created a framework to predict the quality of the evolution of solutions produced by GA. David Goldberg further increased their popularity when releasing his book "Genetic Algorithms in Search, Optimization and Machine Learning" (1989) [9]; where he showed how these algorithms could be applied to machine learning.

They are a type of optimization algorithm based on natural selection and Darwinian evolution. The concept of this algorithms is to generate a large number of solutions (or "individuals"); these individuals are then tested to calculate how close to the best solution each of them is; and after that, a new generation of individuals is created based on the previous generation and their corresponding fitnesses.

### 2.1.1 Genes and Individuals

To understand what an *"individual"* is, it is necessary to understand what a *"gene"* is. A gene is a unit of data in a genetic algorithm, it can contain any kind of data. This includes numbers, characters, strings or other data structures.

An individual represents a solution to the optimization problem that needs to be solved. It is composed by a set of genes and has a defined structure. A simple example of this is an individual that represents a word. In this example an individual can be composed of an ordered set of genes that represents one letter each.

As mentioned before, the genetic algorithm starts by generating an amount of random individuals. This is done by generating sets of random genes and creating individuals with them (these sets can have an order or other relations). The group of all individuals created is called the first generation of the genetic algorithm.

In order to create a new generation in a genetic algorithm, it has to undergo three processes. These processes are Selection, Crossover and Mutation. These three processes allow us to create new individuals based of the individuals of the current population, in order to create a new generation of individuals.

### 2.1.2 Fitness Functions

Let $f : I \to \mathbb{R}$ be a function to optimize, where $I$ is the set of all possible solutions to the problem to solve, this function will be called the "fitness" function. As each individual represents a single solution to the problem, the fitness function can be used to rank these individuals. In genetic algorithms the fitness function is used to calculate how close to the best solution or how good is the solution an individual represents.

This function depends on the problem to solve and thus a fitness function that works well for solving a problem may be terrible for another. However, there are techniques to design fitness functions for some cases and some of them will be covered later in this chapter.

After generating the first generation of the algorithm, all individuals are ranked using the fitness function. The best ranked individuals represent the best solutions the genetic algorithm has at this point and are going to be key in order to generate better solutions. Next, individuals in the population will be selected in order to create new solutions to the problem based on them; this process is called **Selection**.

### 2.1.3 Selection

*Selection Algorithms* are algorithms that take the current population of individuals of the genetic algorithm and return a single individual. The idea behind doing this is to choose individuals in a smart manner in order to generate new solutions to the problem. There are many algorithms to do selection, the one used in this thesis is the Tournament Selection Algorithm (created by Anne Brindle [2]).

This algorithm takes a number of individuals $k$ from the population and returns an ordered array $a$ of $k$ individuals from more to less fitness. Then the elements of $a$ are traversed in order from best to worst. For each element traversed, the element can be elected as the result of the algorithm with a fixed probability $p$; otherwise the algorithm will continue with the next element (the last element in $a$ will be elected with probability 1 if no other is elected before it).

This algorithm is run multiple times in order to select pairs of individuals to create new individuals from their genes. This is done until there are enough pairs of individuals (or parents) to populate the next generation of the genetic algorithm. Each one of these pairs is then used to create a new individual, this process is called **Crossover**.

### 2.1.4 Crossover

*Crossover Algorithms* are algorithms that take two parent individuals and return a new individual based of the genes of the parents. These algorithms depend greatly on the type of the parent individuals. However some standard crossover algorithms exist.

These algorithms are made for individuals whose structure consist of an ordered sequence of genes. The crossover algorithms implemented in this thesis are all based of two on these crossover algorithms. The algorithms in question are the Simple Point Crossover [Holland 1975, De Jong 1975] and K-Point Crossover Algorithms [De Jong 1975] [5].

#### Single Point Crossover

Given two individuals $I_1, I_2$ with the same amount of genes $n$, and given $k \in [1, n]$ a random whole number. The genes of the resulting individual $I$ from performing single point crossover over $I_1$ and $I_2$ are the first $k$ genes of $I_1$ and the last $n - k$ genes from $I_2$ in order (Figure 2.1).

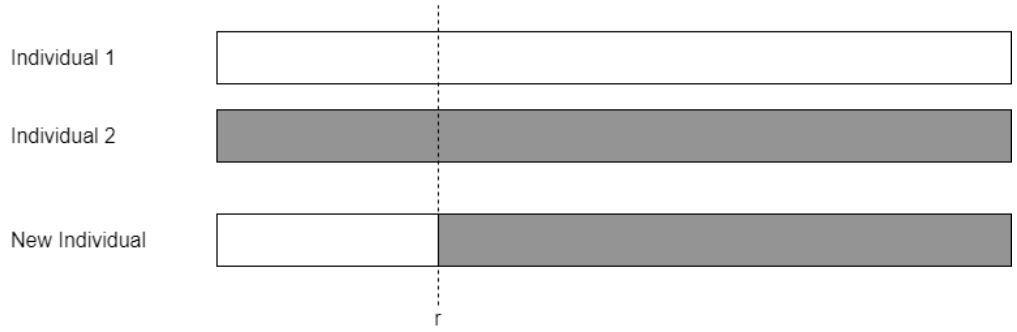


Figure 2.1: A graphic representation of how the genes of two individuals are used to create a new individual using Single Point Crossover.

#### K-Point Crossover

This crossover algorithm is a generalization of Single Point Crossover. Given two individuals $I_1, I_2$ with the same amount of genes $n$, and given $k$ numbers $r_{\mathrm{i}} \in [1, n]$ ($\mathrm{i} \in [1, k]$). Let $I[\mathrm{i}]$ be the *i'th* gene of individual $I$, the genes of the resulting individual $I$ of performing k point crossover over $I_1$ and $I_2$ are given by the following formula:

$$I[\mathrm{i}] = \begin{cases} I_1[\mathrm{i}] & \mathrm{i} < r_1 \\ I_{2-(a \bmod(2))}[\mathrm{i}] & r_{a-1} < \mathrm{i} < r_a \\ I_{2-((k+1) \bmod(2))}[\mathrm{i}] & r_k < \mathrm{i} \end{cases}$$

Where $a$ is a whole number. This result is illustrated in Figure 2.2.

### 2.1.5 Mutation

Previously it was mentioned that there exist three processes that make the GA *"evolve"*; Selection, Crossover and Mutation. Mutation is a process that has a chance to occur every time crossover is performed to generate an individual. In most genetic algorithms, this chance

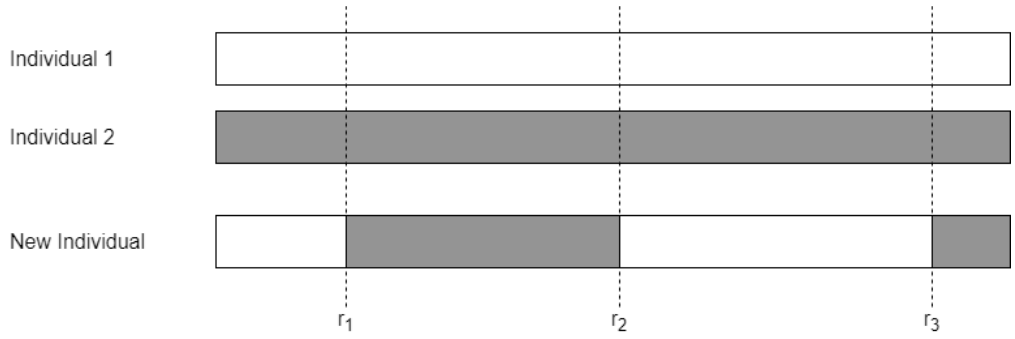


Figure 2.2: A graphic representation of how the genes of two individuals are used to create a new individual using 3-Point Crossover.

consist of a constant user defined parameter. This process consist on modifying the individual that was just created in a random manner.

The most typical way of doing this is to choose a random gene of the individual and modifying it. If a gene is a letter of the alphabet for example, the result of the mutation may change it to another random letter. This changes the individual in a small manner so it may get closer to the best solution for the problem.

After performing the Crossover and Mutation processes, a new generation of individuals is obtained. Then the Fitness Computation, Selection and Mutation processes repeat until a stop condition is met (as shown in Figure 6.10). This condition can be achieving a certain amount of fitness, running for a number of generations or other.

### 2.1.6 Elitism

Elitism is an extra process that is not standard for every genetic algorithm. It consist on selecting some individuals from the current population of a GA and passing them as they are to the next generation. This is done so the fitness of a genetic algorithm does not decrease during execution.

A simple way of implementing this process is to select a number $e_r \in [0, 100]$, this number will be the percentage of the population that passes to the next generation. Then order all individuals by fitness and get the best $e_r$ percent of the individuals to pass them directly to the next generation.

### 2.1.7 Multi-Objective Fitness Functions

A *Multi-Objective Fitness Function* is a fitness function composed of two or more fitness functions. These functions must be designed in a way that any relation between the component fitness functions that could harm the quality of the solution is not encouraged. For example when creating an approximation algorithm [1] with genetic algorithms, it is important that the solution returned is close to the correct solution; and it is also important that it takes a small amount of time. But returning a similar output to the correct one is more important than the efficiency of the program, so its not as simple as to sum both of the fitness functions.

```
if (f(x1) > f(x2)) or ((f(x1) = f(x2)) and (g(x1) > g(x2))) then
    h(x1) > h(x2)
end

if (f(x1) = f(x2)) and (g(x1) = g(x2)) then
    h(x1) = h(x2)
end
```

Figure 2.3: Pseudo code that exemplifies the relationship of the two sub-functions $f$ and $g$ with the fitness function $h$.

Because of these kind of problems, certain techniques have been developed to design multi-objective functions based on relations between the "sub-fitness function" [19]. Some of these techniques are:

### Different Priority Fitness

This technique is used when its possible to assign a "factor of importance" to the sub-fitness functions. For example, let $f, g$ be sub-fitness functions from a fitness function $h$. If its known that $f$ is $k \in \mathbb{R}$ times more important than $g$, then a possible definition of $h$ can be as follows:

$$h(x) = kf(x) + g(x)$$

The value of $k$ can also be estimated based on heuristics.

### Equal Priority Fitness

When optimizing two fitness functions of equal priority, it may be important that one does not increase at the expense of the other decreasing. For this problem a simple technique was created so that every time fitness improves neither of the sub-fitness functions has decreased. Let $f, g$ be sub-fitness functions from a fitness function $h$ with the relation mentioned; then $h$ can be defined as follows:

$$h(x) = min(f(x), g(x))$$

### Prioritized Fitness

Let $f, g$ be *sub-fitness functions* from a fitness function $h$. Lets say that $f$ is more important $g$ so that when comparing two individuals only when they are equal in $f$ we use $g$ to differentiate them.

The easiest way to implement this multi-objective fitness function is letting the output be in $\mathbb{R}^2$. Then its possible to define an order relationship that does exactly the same as the pseudo-code shown in Figure 2.3. There are other ways to represent this relation in $\mathbb{R}$ if some assumptions are taken on $f$ and $g$. Let $Rec(f) = \mathbb{Z}$, using this assumption then $h$ can be defined as shown in Figure 2.4.

$$h(x) = f(x) + sigmoid(g(x))$$

Figure 2.4: The definition of $h$ for a prioritized fitness multi-objective fitness function. The definition of the sigmoid function can be found in the appendix 6.9.

## 2.2 Technologies Used

### 2.2.1 Python

Python [7] is an interpreted, high-level, general-purpose programming language. Created by Guido van Rossum and first released in 1991, Python's design philosophy emphasizes code readability with its notable use of significant whitespace. Its language constructs and object-oriented approach aim to help programmers write clear, logical code for small-scale projects.

### 2.2.2 Lua

Lua [3] is a scripting programming language commonly used to extend or add functionality to software applications. It was designed to be simple and portable. Thus this language has a wide variety of uses but it is commonly used as an extension language for web applications and video games.

### 2.2.3 LÖVE

LÖVE [14] (or LÖVE2D) is an open-source cross-platform framework for developing 2D games for desktop and mobile devices. It is designed in C++ and it uses Lua as a programming language. It is published under the zlib license.

The API provides access to video and audio functions of the host functions through various libraries like SDL [12] and OPENGL [10]. These actions include playing sounds, file management, drawing images and forms, using shaders, etc. The API is built to be simple to use.

# Chapter 3

# Problem

## 3.1 Hard-to-Reproduce Issues

Testing video games is a complex task as they are graphic applications that receive inputs in real time. For this reason the most common way of testing a game in search of issues and errors is by having a group of human testers to play the game. These testers, guided by the QA team check various parts of the game testing new features in search for issues.

However testing the whole game each time a new build is created does not scale well. Thus automated testing techniques like unit testing have been used in recent years as complementary methods to test simple behavior in these applications. This way, testers can focus on testing more complex interactions while the automated testing takes care of more simple actions like equipping an item or triggering some dialog.

Yet there are tasks that are very hard or tedious for human testers. A common task of this kind is doing the same thing with slight variations over long periods of time, for example, attacking an enemy in a shooter with every weapon programmed in the game.

Another task may be accidentally encountering an issue that the tester cannot manage to reproduce afterwards. Little information might be known about the cause of the error, as it was encountered accidentally. Because of this, it can be hard to make a detailed report for the development team to review it and patch the issue.

There are various reasons as to why issues like this may be hard to reproduce: frame-precise inputs may be a requirement for the issue to occur; the issue might depend on a random number generator or computation time; or the issue may be caused because of the order of threads of an internal parallel task of the game. A computer software may be more effective than a human tester when finding these kind of problems.

## 3.2 Solution Requirements

As the objective of this thesis is to apply genetic algorithms in order to solve the problems mentioned before; the following requirements are designed taking into account that a developer will use these tools to debug a game in the future. A genetic algorithm researcher may also use the tools and extend and polish the solution created.

Because of these reasons, there is no defined acceptance criteria for the software. The objective characteristics of this piece of software is to be useful and as easy to use as possible for developers and researchers.

### 3.2.1 Basic Requirements

The software developed in this thesis must have two essential features to tackle the previous problems. The solution developed must be able to **run a sequence of precise inputs on a video game developed using the LÖVE framework**. And it must also be able to **generate precise input sequences based on a problem**.

The second feature implies that the software will need access to information about the problem being tackled. This can be done by **generating a report with in-game variables after running an input sequence**. The report must contain relevant information of the problem to solve, so the software can test input sequences and iterate over them.

These features alone are enough for reproducing frame-precise inputs in games that have a fixed frame-rate and no random or time events. However a lot of games do have these kind of events, to solve this, the software will have to run the game in a way that simulates a fixed frame-rate game with no events as mentioned before. Thus, **the software must set the time elapsed during each frame and the random seed to a fixed value before running a game**.

These two adjustments will make most simple games to run in a deterministic manner. Nonetheless there are particularities in some video games that are too complex to make deterministic. Some examples of these particularities are: online features, rendering-dependant events, os-call-dependant events, non deterministic third party software, etc. These last particularities are rarer to see in video games and are very complex to tackle, so the solution is not required to solve them.

As this software will be used for testing purposes, it must be able to **save input sequences as files** in order to run them as many times as needed. This requirement is included so the development team of a game can replay and analyze an input sequence that produces an error.

Genetic Algorithms were used for the input sequence generation process, as this is the approach proposed in this thesis. Because of this, a set of requirements for the usability of the genetic algorithm must be added.

First, each time the genetic algorithm is run, a report of the parameters and results must be saved as a backup. This is because when running an experiment for testing, most of the

time this data needs to be recorded in order to be analyzed later. So it is useful for the final user that a registry exists for an experiment they did.

The user must be able to **implement their own fitness functions to be used in the genetic algorithm**. As every problem to solve is different, the user must be able to add their own fitness function into the system created, in order to orient the genetic algorithm to get the user's target behaviour.

### 3.2.2 Quality Requirements

The software produced must be able to **run an input sequence in real time**. It should also take as little time as possible to generate an input sequence, as time is very valuable during development. Thus a requirement for this software will be that **it can not take more than one day to produce some meaningful results** in reproducing an issue.

This last requirement depends on the architecture tested. The architecture that will be used for the experiments in this thesis will be a Lenovo Legion Y530-15ICH notebook. The CPU is an Intel Core i5-8300H with 2.30 GHz (which is Quad-Core); the GPU is an NVIDIA GeForce GTX 1080; and it has 8 GB of RAM.

The software must be configurable in a sense that a user should be able to adjust the parameters for generating input sequences based on their problem. Some examples of the wrapper's parameters are: the random seed used and time elapsed between frames. Examples of parameters for the genetic algorithm software are: amount of generations to run, what encoding to use, specific hyper-parameters for encoding creation, etc.

### 3.2.3 Desirable Features

A desirable feature is custom metric computation, user defined functions that are computed alongside the report created by the software, which are not taken into account for generating the solution. This feature would help developers compare in game values and parameters to the problem parameters when the software generates the input sequences. It is a feature that may help the developers get a deeper understanding of the problem and what parameters are involved in it.

As the input sequences generated by the software may be too complex, it is desirable to implement features that help the developers understand them. A desirable feature to solve this problem is a reduction process that simplifies an input sequence while still producing the problem the issue was generated for. Another solution for this problem is implementing a way to visualize input sequences in a simpler manner so developers can understand them better.

It is desirable that the genetic algorithm can be extended so that other people can further research in applying this technology to video game testing. This includes the possibility of creating selection, crossover, mutation algorithms that work in the system. Also included in this feature is the possible extension of current genes and individuals and the implementation of new ones.

# Chapter 4

# Solution

The software we developed for this thesis is composed of two parts. The first part is the *LÖVE Wrapper*, which is responsible of taking sequences of inputs and running them on top of the game. The second part is the *Genetic Algorithm Software*, whose role is to manage the genetic algorithm to generate input sequences; it uses the wrapper to calculate fitness functions.

## 4.1 LÖVE Wrapper

The *LÖVE Wrapper* is a program created in Lua to run input sequences over games made with the LÖVE framework. To do this, the program overwrites the functions the framework uses to read inputs. It also sets the time elapsed between frames and the random seed to a constant value. This is done so the behavior of a game is deterministic and the result of running a specific input sequence is always the same.

It is important to mention that the wrapper's features were implemented to run over the games tested during the development of this thesis. Thus, the wrapper does not re-implement every single method of the LÖVE framework. It just re-implements the methods necessary for it to run over the games tested, and thus events like mouse clicks or touch controls work as normal.

### 4.1.1 The Wrapper's Pipeline

#### Inputs

The Wrapper takes four inputs: the *input sequence file's path*, the *frames to skip*, the *frames to test* and *the frame interval*. The *input sequence file's path* is very self explanatory, it is a string that corresponds to the path of the file where the input sequence to be run is stored.

When calculating the fitness of an input sequence, there is an amount of frames at the beginning that do not count for the calculation as shown in Figure 4.1. These frames are the *frames to skip*, and are left there in case it is necessary to create an input sequence to get to a part of the game before the testing begins. For example, if the fitness function is

defined to test something at the first level of a game, an input sequence can be used to skip the game's menus and get to the important part. Then *frames to test* is an integer that represents the number of frames that will be tested by the fitness function (that come after the frames specified by *frames to skip*).

Finally *the frame interval* is an integer that represents the time between pauses. A pause refers to letting a frame pass internally in the LÖVE framework. Setting this value to 0 will cause that there is never a pause.

This last input is very important because when a frame advances in the LÖVE framework is the only instance the garbage collector is activated. This means that while no frames in the LÖVE framework have passed, the RAM memory usage of program scales continuously with time. That can be a huge problem for longer input sequences, therefore in those cases a positive, integer is necessary for the frame interval. Additionally when a frame passes in the LÖVE framework is also the only instance when the games objects are drawn on the screen.

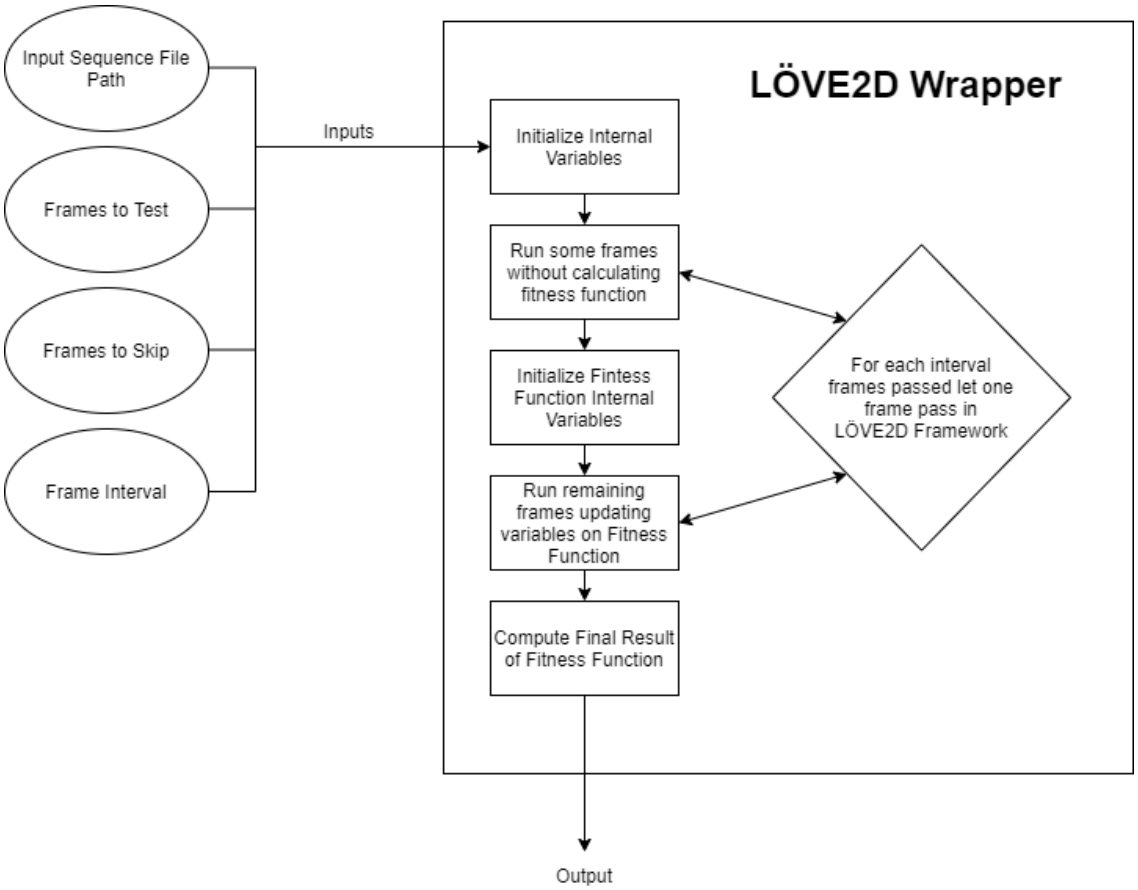


Figure 4.1: The pipeline of the LÖVE Wrapper. It shows the inputs taken by the program and its internal processes.

### Running an Input Sequence over a Game

The wrapper initializes its internal variables given the inputs mentioned before. This process consists of defining a fixed time elapsed per frame; setting the random seed to a constant value; and redefining the random set seed function so no other seeds are set. This way, the

game becomes deterministic and running the same input sequence many times over it will always produce the same results.

After initializing the internal variables, the wrapper advances a number of frames equal to the *frames to skip* input variable. Once every *frame interval* frames has passed, the wrapper will let a frame pass in the LÖVE framework. These frames are not considered in the fitness computation.

Once these frames had advanced, the wrapper will initialize the fitness function variables. Then it will advance a number of frames equal to *frames to test* while making fitness calculations every frame. Finally, after advancing those frames, the final computation of the fitness function will be made and returned in a json dictionary through the standard output. The fitness function processes are explained in more detail in Section 4.1.3

It is important to note that the wrapper does not block user inputs while it is running. Because of how the framework works, if a game overwrites some of its events (specifically the ones that check for inputs), then the wrapper must use the new processes for the events defined. Because of this, these events can be triggered during execution.

This means for example, that if the user presses the "k" key while the wrapper is running, the framework's events will trigger as normal. So no keyboard inputs should be passed to the wrapper during execution.

#### Metric Computation

A more advanced feature of the wrapper is that it allows for multiple values to be computed alongside the fitness function. These other values are computed by functions that work in the same manner as fitness functions do. From this point on, these values will be referred to as metrics, as they represent characteristics of the execution of an input sequence, not necessarily used for fitness computation.

In the dictionary returned by the wrapper, the fitness value is associated to the key *fitness*. The other contents of the dictionary are the metric values, and each metric is associated to a user-defined key.

Metrics correspond to an important feature of the wrapper. This is because these values can be compared alongside the fitness function to reach some conclusions about the design of said function. Examples of metrics can be: memory used, final distance of the player character to a point, etc.

### 4.1.2 Input Sequence Encoding

Input sequences are represented as a dictionary whose keys are integers that correspond to a frame during the game's execution. The values are dictionaries that represent keyboard key inputs presses and releases on that frame.

In the dictionary in Figure 4.2, the keys represent the keyboard keys (*space* is the space key and *up* is the up arrow key), the keys' strings are the same used by the LÖVE framework. The

```
{
    "space" : false ,
    "up" : true
}
```

Figure 4.2: An example of a dictionary associated to a frame of an input sequence file.

values of the dictionary are booleans that correspond to an input's activation or deactivation (*true* for activation and *false* for deactivation). Thus the dictionary presented before, specifies that the *space* key is being released and the *up* key is being pressed in the frame portrayed.

The encoding of the input sequences correspond to Python AST dictionaries [18], which are similar to json dictionaries. The difference between valid json files and the encoding of these input sequences, is that json files do not use numbers as keys for dictionaries. Figure 4.3 presents an example of a valid input sequence.

```
{
    3: {
        "space" : true
    }
    6: {
        "space" : false ,
        "up" : true
    }
}
```

Figure 4.3: A valid input sequence that represents the space key being pressed on frame 3, and then on frame 6 the space key is released and the up arrow key is pressed.

### 4.1.3 Fitness Functions

The Wrapper is in charge of computing the fitness function for an input sequence. The fitness function is represented as a class internally by the program. This class has three important methods that are called by the wrapper: the initAux method, the mainFun and the stepFun method.

#### initAux Method

The initAux method initializes the internal variables of the fitness function. This method is run the moment before the frames to be tested start. For example, if the fitness function measures the difference in memory usage over the tested frames, this method will store the current memory usage in order to compare it with the final usage.

#### mainFun Method

The mainFun method returns the final fitness of the input sequence. As this implies, this method is run when all frames to be tested have passed. In the example of the fitness function

that measures the difference in memory used, this method would return the difference between the current memory used and the memory used calculated in the initAux method.

#### stepFun Method

The *stepFun* method is run once every frame during the frames to test. This method is used to update the internal variables of the fitness function every frame. It allows for more complex fitness functions to be created, by allowing the user to collect data every frame and in the last frame make computations over all the data collected.

For example, lets say the objective of the fitness function is that a player character moves as much as possible. This means it is insufficient to check just the starting and ending point of the character, because if a player moves in circles, the total amount of movement can not be represented with just the start and end positions. The *stepFun* method can be used to calculate the position of the player character in each frame and in the *mainFun* method the total distance the player moved can be calculated with the data obtained.

## 4.2 Genetic Algorithm Software

This software is a program made in Python that runs genetic algorithms over input sequences and tests them using the LÖVE Wrapper. To do this, the program encodes input sequences as an individual with a sequence of genes. There are many ways this can be done, depending on how the genes and individuals are defined.

### 4.2.1 Genes

During the development of the software, a variety of gene types were implemented. These types were divided into two types: **dependent genes** and **independent genes**.

**Dependent Genes**, as the name suggests, are genes that depend on an external structure (provided by an individual) to create an input sequence. This structure can be just giving an order to a sequence of genes. For example a gene may represent a key press and the individual composed of genes from this gene type can represent a sequence of key presses in order (given by an order on the genes).

**Independent genes**, on the other hand do not depend on an external structure to create a valid input sequence. Genes of this type represent a part of the final input sequence. An example of this can be a gene that represents a 1-frame key press and a frame when the key is pressed. An individual made from genes like this will just serve as a collection of genes.

The type of genes implemented during this thesis are the following:

- **Single Input Key Gene:** A dependent gene that represents a single key press.
- **Key Frame Gene:** An independent gene that represents: a 1-frame key press and the frame the key press was made.
- **Key Frame Duration Gene:** An independent gene that represents a key press, along with the frame the key will start to get pressed and the duration in frames the key will

continue to be pressed.

- **Ephemeral Key Gene:** A dependent gene that represents a key press and the number of frames the key will continue to be pressed. A variation of this kind of gene is the **No-Input Ephemeral Key Gene**, an Ephemeral Key Gene that has a null input associated to it.

### 4.2.2 Individuals

An individual represents an input sequence is composed of a set of genes. In this thesis two types of individuals were implemented: individuals that can represent simultaneous inputs and those that can not.

There are fitness functions where only input sequences with simultaneous inputs will get good results. However this kind of expression power is not always necessary. Restricting the individuals to input sequences that do not have multiple simultaneous key presses will reduce the size of the space of the solution; and thus the genetic algorithm will converge faster.

For every individual, some extension of Single Point and K-Point Crossover was implemented. Many individuals also have a variety of mutation functions associated, aside from the classic one that consists of just mutating one gene. The individuals implemented in this thesis are:

#### Input Key Individual

This is an individual composed by only genes of the type *Single Input Key Gene*. It represents a sequence of inputs (each one lasting one frame) that are executed in order.

#### Input Key Frame Individual

This is an individual composed by only genes of the type *Key Frame Gene*. Each gene of this individual represents a specific 1-frame key press in the input sequence.

For this individual, a different version of Single Point and K-Point Crossover was used. As two individuals may have different amounts of genes for an input sequence of the same length, these crossover functions are performed as they would work in an Input Key Individual. For example in simple point crossover, the random cut represents a frame, and all inputs performed before it from the first parent are passed to the new individual; as well as the inputs of the second parent performed during and after that frame.

#### Input Key Frame Duration Individual

This individual is the same as the *Input Key Frame Individual* but has *Key Frame Duration Genes* instead. As this individual is an extension of the Input Key Frame Individual it also inherits that individual's crossover functions.

#### Ephemeral Key Individual

This individual can be composed of genes of the *Single Input Key Gene* and *Ephemeral Key Gene* types. It represents a sequence of key presses in order where the single input key genes

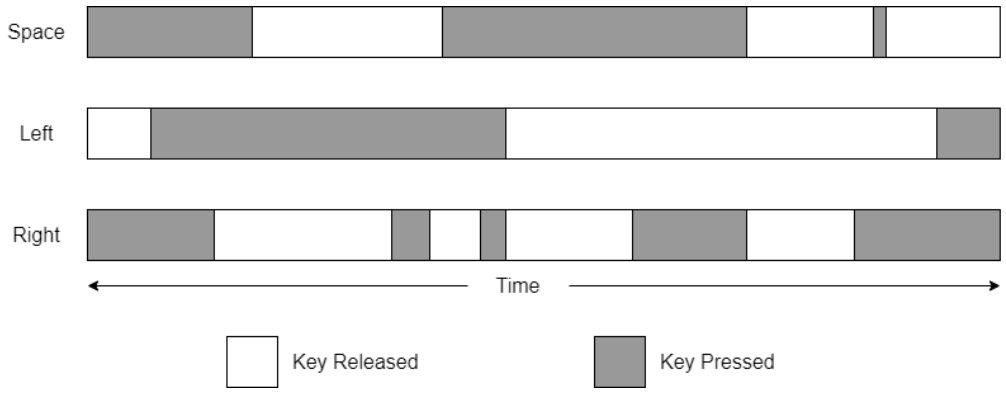


Figure 4.4: A visual representation on how the Multi Ephemeral Key Individual represents an input sequence. Each area (both dark and white) of the rectangles represent an ephemeral key gene.

last one frame. The other types of genes have information on how many frames they last.

#### Multi Ephemeral Key Individual

This individual is composed of just genes from the *Ephemeral Key Gene* type. It groups the genes by input and for each of these groups generates an input sequence. Each of these sequences is composed of times when the input is pressed and times when the input is not pressed, given by the duration of the ephemeral key genes (Figure 4.4). Finally all these input sequences are merged. This merge action is performed just by overlapping all input sequences generated.

Here the Single Point and K-Point Crossover is performed in the same way as Input Key Frame Individuals. This means that for example in the case of Single Point Crossover a random cut represents a frame; then the new individual is constructed so that it performs the same as their first parent before the random cut and the same as their second parent during and after the random cut. Also, for this individual in particular, the Uniform Crossover operation does not make sense so it is not implemented.

### 4.2.3 Testers

A tester in this software is an object that takes an individual and returns its fitness. The tester is therefore used by the software when obtaining the current generation's individuals fitness. To do this, the tester turns individuals into an input sequence files, runs the LÖVE Wrapper with said files and returns the fitness values.

When calculating the fitness of an individual, the tester must first erase all the external data associated to the game. To do this, an user-defined input sequence is run. This input sequence is run over the game to delete the data, by deleting save files for example.

Next, the tester saves the individual as an input sequence file and uses its path as input to the wrapper. The tester has a set number of seconds to timeout (60 by default), when the timeout is hit (which can be from a game crash or just that the testing took too long), the fitness value is set to a low value (-9,999,999 by default) and the input sequence generated is

instantly saved in the output directory. Thus the software continues to be executed until it finishes the end condition.

### 4.2.4 Timeline

During the span of the project, the software developed went through many iterations. In the beginning, the software developed was meant to be one executable program that would generate and run input sequences. However, because of technical complications when trying to reset the memory used by the game between input sequence runs; the software was divided later into two pieces of software.

The original methodology for the development and validation for this software can be seen in the Gantt Chart in Figure 4.5. As it can be seen in the Figure, from August to mid October 2019 the first version of the wrapper was going to be developed. After that, the software was going to be tested using luaAsteroids until mid November, when some updates and upgrades were supposed to be made to the software in case some problems arose, until late December. Finally, March to June 2020 was the testing and validation period, and a number of games would be tested and small modifications were supposed to have been made during this period.

Because of external issues and the technical complications mentioned before, there was a span of time between October and November where no progress could be achieved. During the month of December a new Gantt Chart was designed in order to catch up with the final deadline (Figure 4.6).

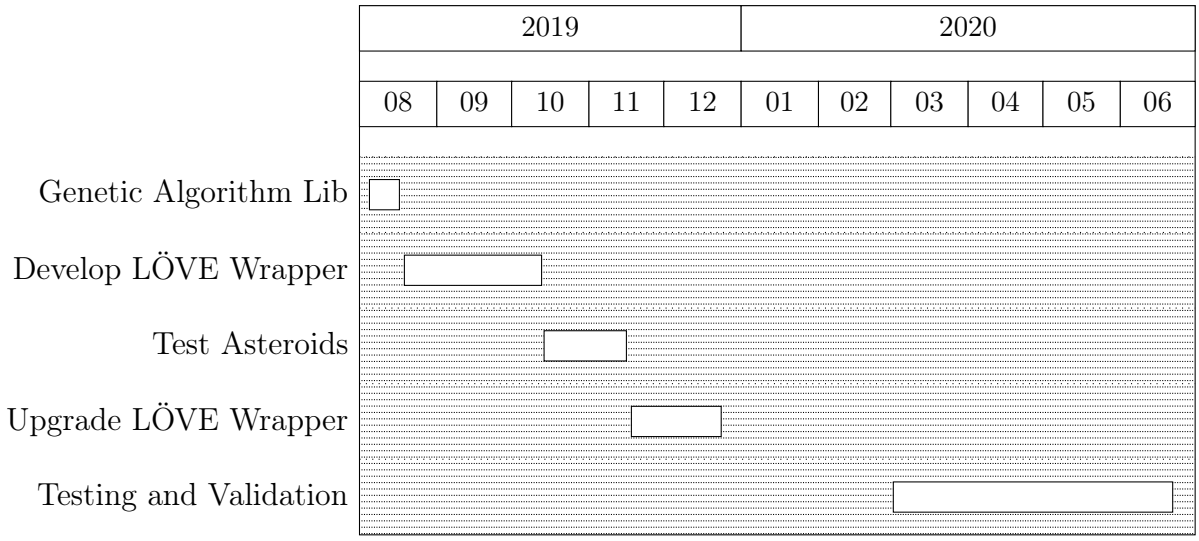


Figure 4.5: Gantt chart of the methodology designed at the beginning of the project.

The Gantt Chart shown in Figure 4.6 resulted to be very accurate on the times of the work done. The structure of the genetic algorithm to be used was developed during August. During the second half of August until mid October the wrapper and the genetic algorithm were re-implemented as one software.

During the *Test Asteroids Phase* the Input Key Individual, Input Key Frame Individual,

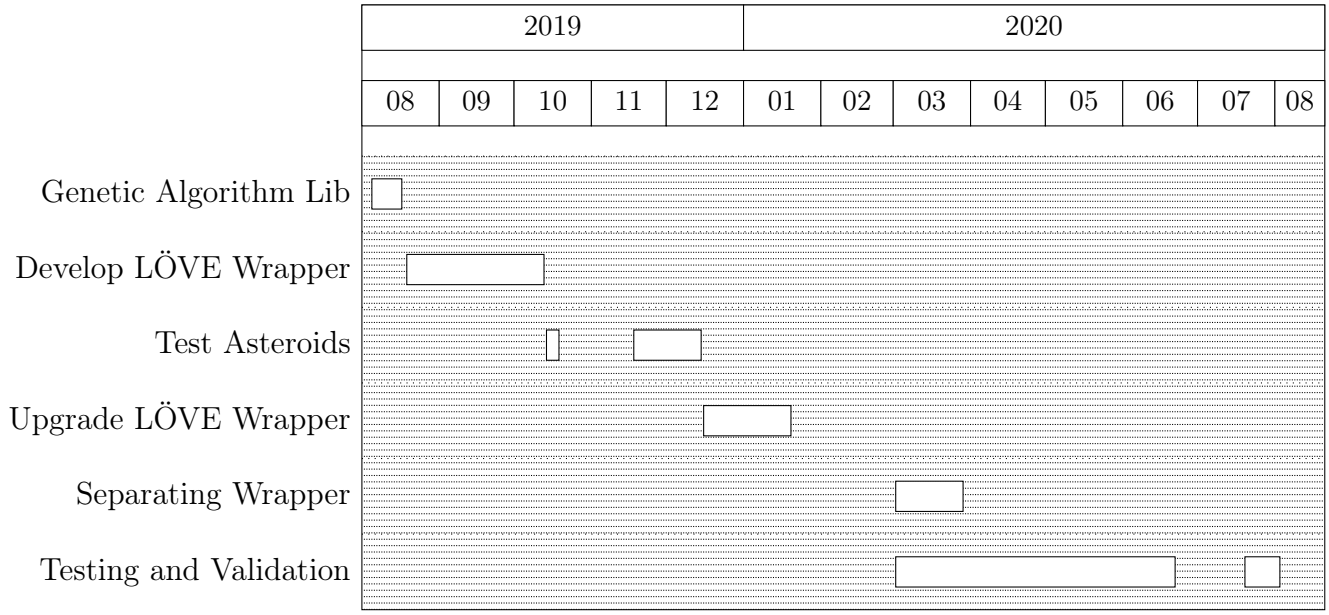


Figure 4.6: Gantt Chart of the methodology designed in December 2019 after the time interval in which no work could be done.

Input Key Frame Duration Individual and the Ephemeral Key Individual were developed. These encodings were compared in order to test performance and efficiency when reproducing an issue. More details of the experiment done during this phase can be found in section 5.1.

The *Upgrade LÖVE Wrapper* phase consisted on developing the Multi Ephemeral Key Individual and preparing for testing new games. These preparations included the features for skipping menus, the development of the fitness function structure and the frame interval.

At the start of the testing and validation phase, the software was separated in two pieces because of complications when attempting to run the second game to be tested. This included a complete re-implementation of the genetic algorithm and the encodings. When doing the re-implementation, the code was structured in a more modular way so the encodings and algorithms could be extended easily.

During the rest of the testing and validation phase, we selected some issues from the game *Journey to the Center of Hawkthorn* to validate the software. We also contacted the developers of *Zabuyaki*, who explained some issues they needed to know more information about. In order to test *Journey to the Center of Hawkthorn* specifically, a special tester was developed to speed up the testing process, and an input sequence was implemented to clean save data before the testing of an individual.

Finally at the end of this phase (in the last weeks of July), we made some experiments to measure the execution time of the software. These experiments measured metrics like testing and reproduction time during software execution in order to approximate how much time the software would take in a general case. We additionally performed some experiments to test some features that accelerate the execution time of the software.

# Chapter 5

# Validation

We tested three games in order to validate these tools: LuaAsteroids, Journey to the Center of Hawkthorn and Zabuyaki. For each game, we identified some issues by playing the game and searching online. This section presents a number of experiments to test issues from each game; following with a few experiments oriented to test the software's execution time.

## 5.1 LuaAsteroids

This first game tested is heavily inspired by the classic game Asteroids, this implementation was created by Lang-CS and posted in Github. It is basically a re-implementation of the collision mechanics of the game, it generates asteroids from the border of the screen going to the center. The player can move the same way as in the original game and shoot to destroy the asteroids. The asteroids do not break into smaller asteroids when they are hit, but are instead destroyed.

This game has a memory leak the developers are aware of, and its cause is that the asteroids are not destroyed after they are no longer visible on the screen. Because of the simple nature of the game and the necessity to test the encodings developed until that moment, this game was used to get metrics from the encodings created.

For the experiments regarding this game, the fitness function used returned the multiplicative inverse of the difference in RAM memory between the start and the end of the execution (Figure 5.1). As the genetic algorithm maximizes the fitness function taken, it is expected that the genetic algorithm destroys as many asteroids as possible in order to free memory used by the asteroids that spawn. The variables used by the genetic algorithm when run are the following:

- **Generations:** 30
- **Individuals by Generation:** 100
- **Mutation Probability:** 0.05
- **Elitism Ratio:** 0.05
- **Frames to Test:** 500

$$fitness = startRAM - endRAM$$

Figure 5.1: Fitness Function used by the genetic algorithm on order to see the performance of each encoding.

The following encodings were tested in the experiments:

- **Encoding #1** corresponds to an *Input Key Individual* with an amount of genes equal to the amount of frames tested. Each one of the genes were generated randomly.
- **Encoding #2** corresponds to an *Input Key Frame Individual*. The amount of genes was varied throughout the experiment between the values: 20, 40, 80 and 160. These genes were generated randomly in frames between the first and last frame to test.
- **Encoding #3** corresponds to an *Input Key Frame Duration Individual*. The amount of genes was varied between the values: 20, 40 and 80. The maximum duration of these genes had when created varied between: 5, 10 and 20 frames.
- **Encoding #4** corresponds to an *Ephemeral Key Individual* with just *Ephemeral Key Genes*. The amount of genes was varied between the values: 20, 40, 80 and 160.

  The duration in frames of these genes was generated randomly using a uniform random variable. This variable has *"1"* as the minimum value, and the maximum *"k"* is defined so the expected frames that the sequence generated lasts is the amount of frames to test. This value is defined as:

  $$k = \lfloor \frac{2(frames\_to\_test)}{(amount\_of\_genes)} \rfloor$$
- **Encoding #5** corresponds to an *Ephemeral Key Individual* with *Single Input Key Genes* and *No-Input Ephemeral Key Genes*. For this encoding, the values that varied were the ratio of *No-Input Ephemeral Key Genes* with respect to the total, and the maximum duration of these same genes. The ratio values were 5%, 10% and 20%; and the maximum duration values were 5, 10 and 20 frames

  The total gene amount $g$ was calculated using these values so the expected number of frames the sequence last is the amount of frames tested. Let $r$ be the ratio, $f$ be the frames to test and d the maximum duration, then $g$ is calculated as follows:

  $$g = \lfloor f \times ((1 - r) + \tfrac{2r}{\mathrm{d}}) \rfloor$$

### 5.1.1 Results

Each encoding was tested varying the internal parameters associated with it. Through these experiments, we determined the "best" version of each encoding as the set of hyper-parameters that maximized the encoding's fitness by the end of 30 generations. The in-depth results for these experiments can be observed in the appendix (Figures 6.1, 6.2, 6.3, 6.4, 6.5, 6.6, 6.7 and 6.8). As Encoding #1 did not have any hyper-parameters, there was no need to make a hyper-parameter comparison.

After these experiments, we did an experiment comparing the "best" version of every encoding. The encodings versions for this experiment were the following:

- **Encoding #1:** This encoding has no hyper-parameters, so adjustments are not necessary.

- **Encoding #2:** Gene amount 40.
- **Encoding #3:** Gene amount 40 and max duration in frames 10.
- **Encoding #4:** Gene amount 20.
- **Encoding #5:** Ephemeral gene chance 20% and max ephemeral gene duration in frames 10.

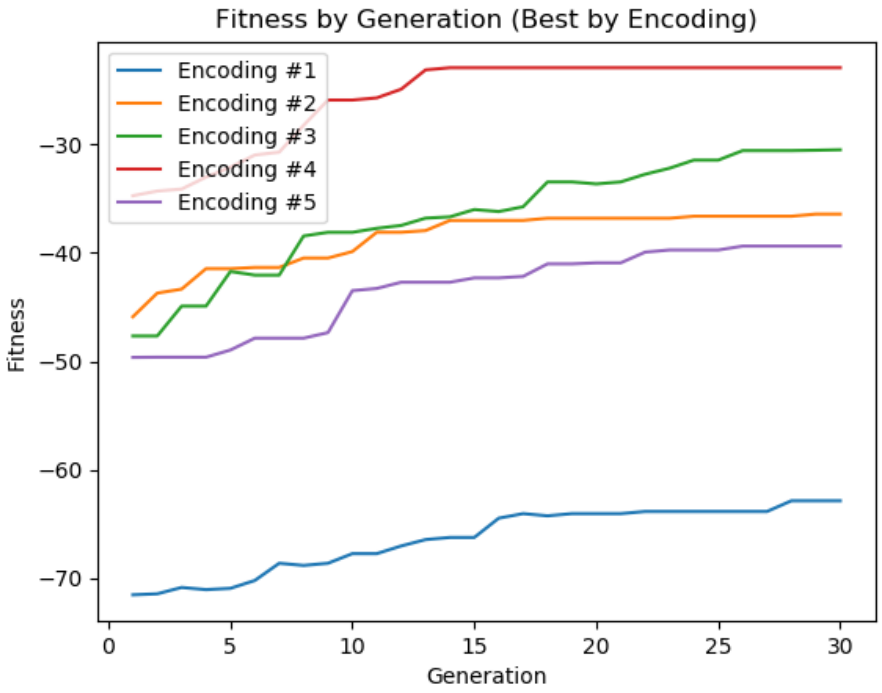


Figure 5.2: Comparison of the best encodings by generation 30 taking in account just the final result.

Looking at Figures 5.2 and 5.3, the first encoding produced the worst results in terms of fitness achieved and reproduction time. Encoding #4 on the other hand presents the best results in both categories, converging very fast and obtaining the highest fitness achieved of the encodings. Encoding #5, as a mix between encodings #1 and #4, seems to have intermediate results between these two encodings.

Encoding #2, composed of *Key Frame Genes*, converged as quickly as encoding #4 and in the first 14 generations it had results similar to #3; the encoding that consists of *Key Frame Duration Genes*. On later generations, encoding #3 seems to grow steadily in fitness and does not appear to be near convergence by generation 30. In terms of reproduction time, encoding #3 takes a couple more milliseconds than #2 but both values are a lot smaller than the ones from encoding #1 and #5.

From these results it seems that the encodings with genes that represent inputs being held for a number of frames have better results in terms of fitness. It also seems that given the criteria used, encoding #4 is the best encoding as it achieves the best fitness and takes the least amount of reproduction time. Because of this, we did more research on encoding #4, leading to the creation of a new individual encoding, the *Multi Ephemeral Key Individuals*.

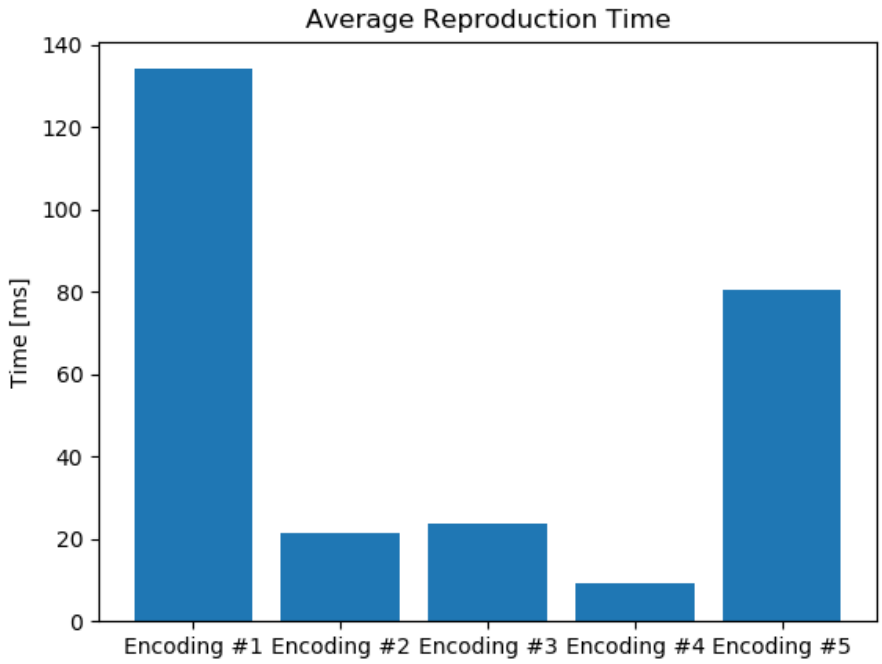


Figure 5.3: Comparison of the best encodings time taking the mean of 30 generations.

## 5.2 Journey to the Center of Hawkthorn

Journey to the Center of Hawkthorn (or Hawkthorn for short) is a side-scroll platformer rpg with many features and systems. The game was developed in 4 years (from 2012 to mid 2016), it has 59 contributors on Github and more than 5000 commits. Between them there is a dialog tree system, crafting system, battle system, etc.

Most of the game is about platforming and battling enemies in thematic levels. On these levels, the player can move left and right, jump, crouch, crawl, open the inventory, collect items, attack, etc.

### 5.2.1 Moving Platform Experiments

Looking at the issues of the game's repository in Github, the moving platforms in some of the levels have a problem. When the player spams the crouch, attack and interact keys they might end up falling through the moving platforms. To replicate this bug we designed the following fitness functions:

- **Stay Fitness Function:** A fitness function that rewards the player for staying in the starting position (especially on the first frames). Given $n$ the number of frames to test; $x_i, y_i$ the coordinates of the player in frame i; and $k \in (0, 1)$. This fitness function is defined by the following formula:

$$fitness = -\Sigma_{i=1}^{n} k^i (\sqrt{(x_0 - x_i)^2 + (y_0 - y_i)^2})$$

- **High Fitness Function:** A fitness function that rewards individuals for staying as


Figure 5.4: The first platform tested in Journey to the Center of Hawkthorn.

high in the $y$ coordinate as possible. This decision was made so the player is rewarded from jumping on the platform. This function is defined by the formula:

$$fitness = \Sigma \mathrm{i} = 1^{n} y_{\mathrm{i}}$$

Using these formulas, the genetic algorithm was tested in three different moving platforms. The first platform (Figure 5.4) is in the upper part of a level called *village-forest*, around the in-game position *(x: 96; y: 23)*, if a player falls, it is difficult to get back up. The second platform (Figure 5.5) is just above a death pit in a level called *village-forest-stonerspeak* around the in-game position *(x: 125; y: 33)*, so falling would mean the player dies. The third platform (Figure 5.6) is under constant enemy fire in the same level as the previous one in position *(x: 165; y: 30)*; because of the enemy fire, it is hard to stay constantly on the platform.

It is worth mentioning that for both fitness functions, the amount of fitness an individual obtains will be reduced by a great amount for each frame the player character spends outside the initial room. This is important as if the player dies (for example in the death pit of site 2), the player character will be transported to another room. This way, dying and exiting the room result in behaviors that are heavily punished.

During the experiments, we noted that the mutation function was making changes too drastic to individuals. After doing the first experiments we made changes to the mutation function. The results presented in the following two experiments are a comparison of those two mutation functions.


Figure 5.5: The second platform tested in Journey to the Center of Hawkthorn.


Figure 5.6: The third platform tested in Journey to the Center of Hawkthorn.

### 5.2.2 Stay Experiment

#### Experiment Parameters

For this experiment, we added a restriction to the input sequences so that the genetic algorithm can not make use of the inventory button for the game. This is because when triggering the inventory in the game, the player's movement is locked and the enemies' AI is paused. That would make the Stay Fitness Function's experiments solution trivial by just pressing the inventory button at the beginning.

The following GA parameters were set for the experiments of both mutation function implementations:

- **Generations:** 30
- **Individuals by Generation:** 100
- **Mutation Probability:** 0.05
- **Elitism Ratio:** 0.05
- **Frames to Test:** 500

We used Ephemeral Key Individuals for this experiment; and the initial population was created so the input sequences generated lasted the tested frame amount. We chose this kind of individuals because they create simple input sequences and staying in a moving platform is a task that can be done without simultaneous inputs.

For the individuals with the old mutation function we set the mutation probability to 5% and for the ones with new mutation function we set it to 20%. This is because the old mutation function produced many drastic changes and the new one produced smaller changes. Thus we decreased the mutation probability when testing with the new mutation function.

#### Results

Looking at Figures 5.7, 5.8 and 5.9, it can be observed that the algorithm with the old mutation function converges too quickly. The three curves that represent the old mutation function have little to no minor improvements, all of the improvements are major. In Figures 5.7 and 5.8 the algorithm converges before generation 6 and in Figure 5.9 the convergence occurs in generation 10.

For the individuals with the new mutation function in the first platform, it took 7 generations for them to stay on the platform for the whole testing time (Figure 5.7). After that, small optimizations can be seen, between them were: re-positioning the player character to be closer to the starting point and removing some jumps. Throughout all the experiment, small improvements were made almost every generation.

For the individuals in the second site, it took 2 generations for them to stay on the platform. In generation 9, the individuals found out that staying in a static platform close to the moving platform gave them a higher fitness score (Figure 5.8). The rest of the generations' improvements were arriving to this static platform sooner and staying close to the ledge to get as close to the starting position as possible.

For the individuals in the third platform, most improvements throughout the generations consisted of dodging the enemy bullets by crouching. As the generations advanced, the individuals adjusted to the timing of the bullets being shot at them, all of this while staying in two moving platforms.

When comparing the individuals in platform 1, the difference between the ones with the old and new mutation function is that the latter ones tend to move a lot more, adjusting their position constantly. In case of platform 2 the individuals from the old mutation function tended to stay in the platform while the individuals with the new mutation function found a static place to stay while optimizing little by little how close they were to the starting point. Finally in case of platform 3 the individuals with the new mutation function moved a lot more between platforms than the individuals with the previous one, adjusting their position and making small optimizations over the generations.

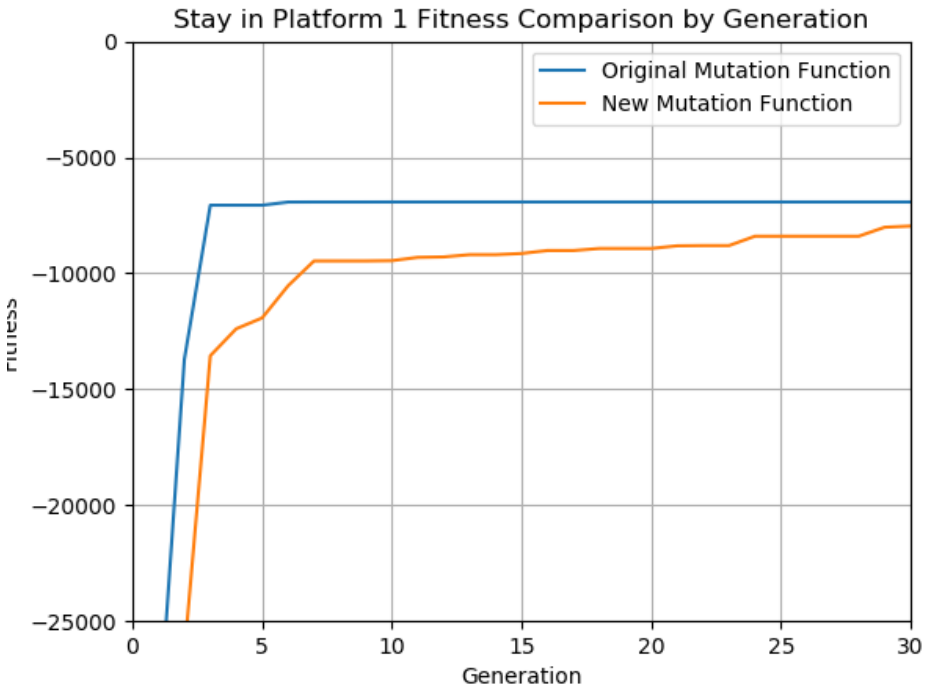


Figure 5.7: Fitness by generation of Hawkthorn's Stay Fitness Function's Experiment on the first platform, comparison between mutation functions.

### Experiment Conclusions

The resulting individuals with the old mutation function tend to stay still for long periods of time. It is believed that this is the reason why the individuals with the old mutation function reach a high fitness so quickly. The drastic nature of the mutation function would make small changes to adjust position really rare, so it would be hard to further improve the generation.

On the other hand the resulting individuals of the genetic algorithm with the new mutation function were constantly making small improvements in fitness through the generations.

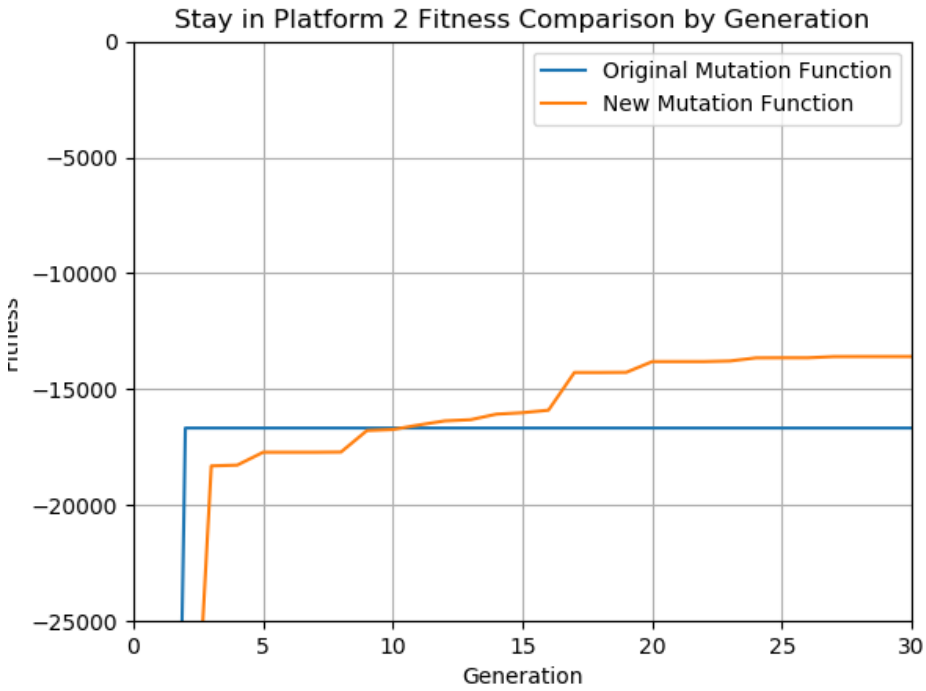


Figure 5.8: Fitness by generation of Hawkthorn's Stay Fitness Function's Experiment on the second platform, comparison between mutation functions.

Because of this, while managing to stay in the platform for more time, the small optimizations made the individuals to polish their movement constantly. This led them to converge later than the individuals with the previous fitness function and in two of the three cases to overcome them in terms of final fitness value.

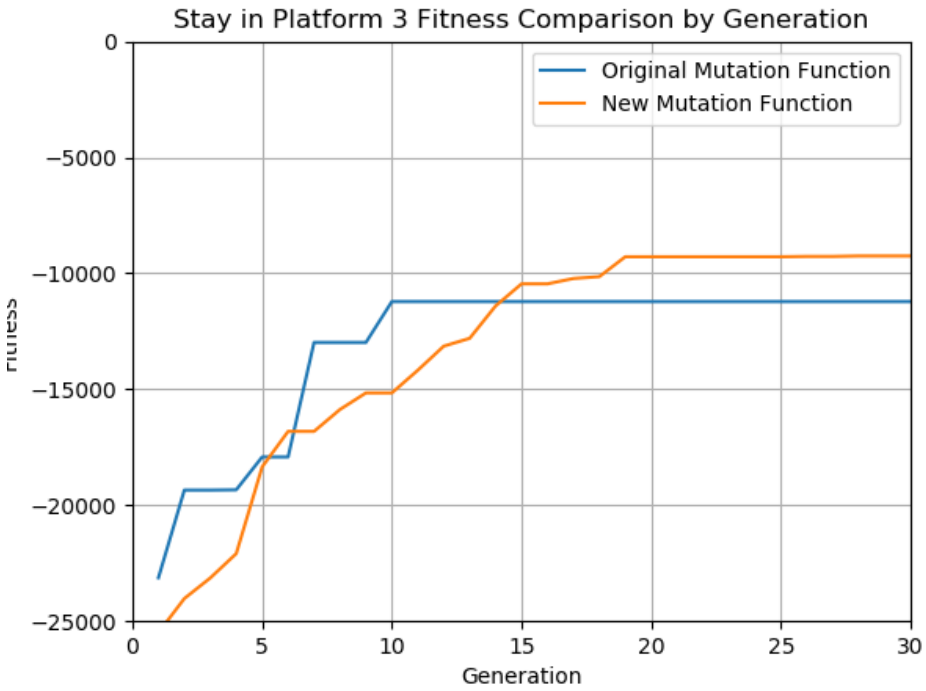


Figure 5.9: Fitness by generation of Hawkthorn's Stay Fitness Function's Experiment on the third platform, comparison between mutation functions.

### 5.2.3 Out of Bounds Experiment

In the game's repository in Github there is an issue called *Collision - Attacking when coming up from crouch* [13]. This issue is about a glitch that causes the player to get inside walls in places where there are sections the player has to crawl to pass (Figure 5.10). It was discovered by LoubiTek in June 2015 and the way of reproducing it was found by niamu also in June 2015. In the discussion surrounding the issue in the github repository, niamu explained that they reproduced the glitch in tight spaces when the player must crawl by crouching under the space and attacking while coming out of crouch as the name of the issue says.

There is a small amount of levels in the game where it is necessary to crawl to navigate. From those there is an even smaller amount that are designed as tight spaces which is where this issue is said to occur. Three of these instances were found: one in the *forest* level, other in the *black-caverns-2* level and one final in the *vents-2* level.

The instance in the *forest* level is around the in-game position *(x: 116, y: 11)* and consists of a short space where the player has to crawl to pass (Figure 5.11). The instance in the *black-caverns-2* level is around the in-game position *(x: 7, y: 6)*, this consists of a moving platform and a small space, the idea is to crouch in the moving platform to pass (Figure 5.12). Finally the instance in the level *vents-2* around in-game position *(x: 25, y:6)* is very similar to the instance in *forest* and can be seen in figure 5.13.

For the experiments to reproduce this glitch, we developed a fitness function that rewards an individual for going as close to an in-game position as possible. For every site we selected a


Figure 5.10: Image of Issue 2456: *Collision - Attacking when coming up from crouch* [13].


Figure 5.11: The first site where the glitch in issue 2456 [13] will be attempted. To advance it is necessary to crawl through the tunnel in front of the player character.

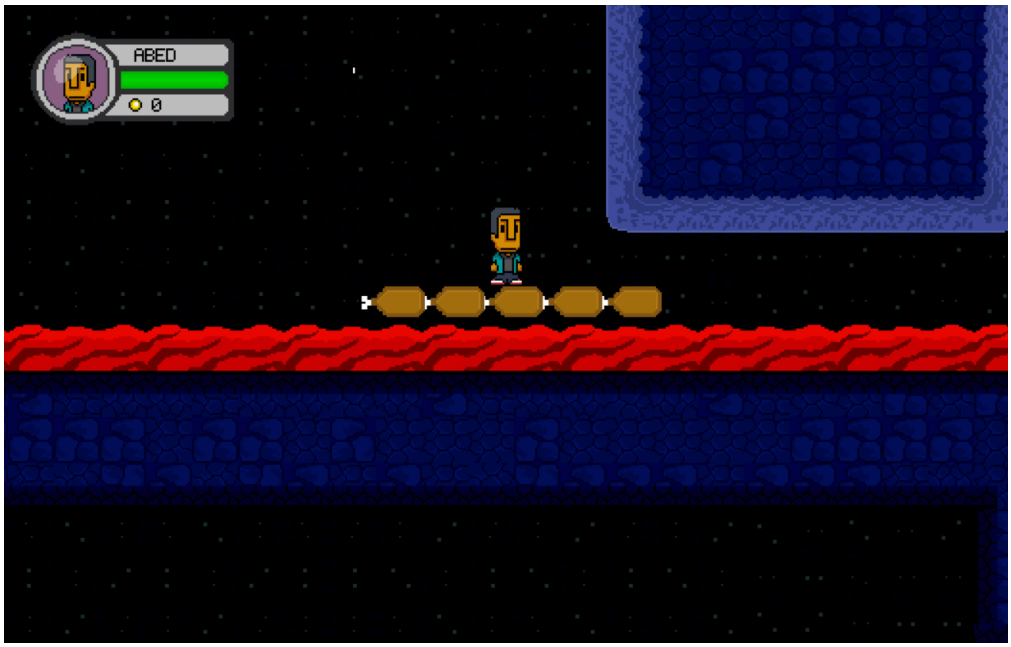


Figure 5.12: The second site where the glitch in issue 2456 [13] will be attempted. In this site in particular it is possible to advance just by crouching.


Figure 5.13: The third site where the glitch in issue 2456 [13] will be attempted. To advance it is necessary to crawl through the tunnel to the left of the player character.

specific spot inside the wall over the crouch-able space as the fitness function target position. We decided to use Multi Ephemeral Key Individuals for this task, as they can represent simultaneous inputs and handle the action of holding inputs particularly well.

### Experiment Parameters

For each experiment we set the genetic algorithm's the parameters as follows:

- **Generations:** 30
- **Individuals by Generation:** 100
- **Mutation Probability:** 0.2
- **Elitism Ratio:** 0.05
- **Frames to Test:** 500

For the second site in particular, we developed an input sequence to wait in the moving platform for a bit before starting to act. We did this so the genetic algorithm does not have to also worry about maintaining itself in the moving platform. This sequence took 125 frames, so the genetic algorithm had 375 frames to act after the skip sequence ended.

We disabled the inventory and pause inputs for these experiments. According to the issue, those inputs were not used when the glitch was produced for the first time. Thus the inputs were disabled because it would be considered noise for trying to reproduce the glitch.

### Results

For the first site, the glitch was not produced; the genetic algorithm instead produced individuals that got as close as possible to the target position by crawling near it. In the second site, the glitch was produced in generation 1; in the following generations the genetic algorithm tended to produce individuals that stayed increasingly closer to the target point inside the wall (Figure 5.15). For the final site, the genetic algorithm spent the first 3 generations jumping against the wall until in generation 4 it performed the glitch; the rest of the generations' improvements were similar to that of the second site experiments, where the genetic algorithm gradually got closer to the exact point inside the wall of the fitness function (Figure 5.16). This last behavior can be observed in the fitness progression in Figure 5.14.

By analysing the data from the experiments, we discovered that in site 2 the resulting individuals crouched; attacked; and shortly after attacking, came out of crouch. In site 3 on the other hand, the resulting individuals accumulated speed and came into crouch, sliding into the gap where it was supposed to crawl and then performed the same actions as the individuals in site 2.

After reducing the output input sequence for the second platform's experiment manually, we created an input sequence of three inputs that produces the glitch. This sequence consists of pressing the crouch button, then when on a space the player character must crouch, press the attack button; and before 4 frames pass, release the crouch button. The glitch causes the player character to go up until it is not collisioning into a collision box, which gets them inside the wall.

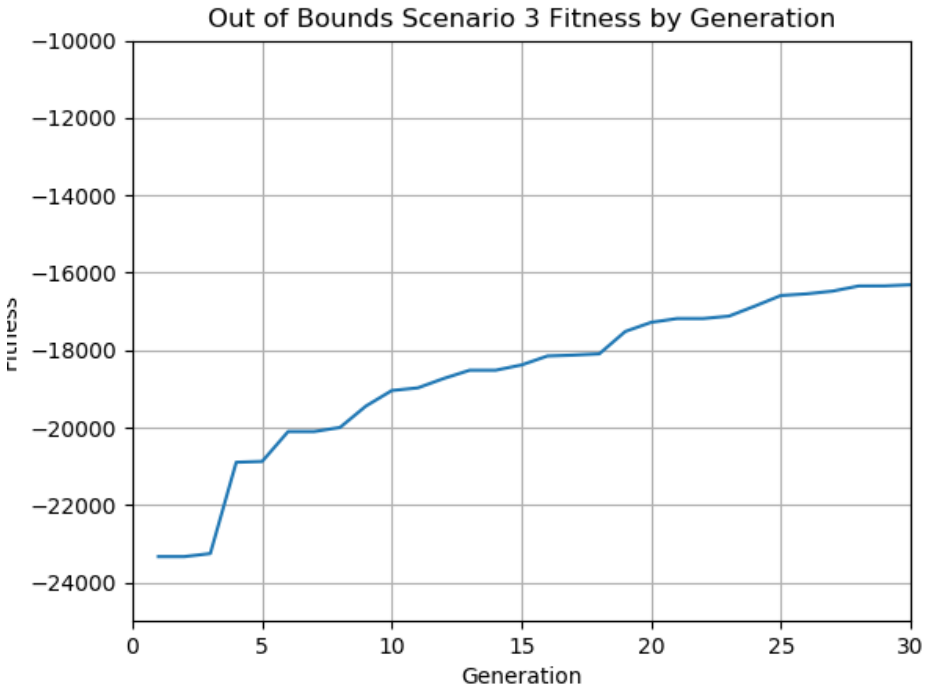


Figure 5.14: Fitness by generation of the genetic algorithm in the third site for the out of bounds glitch.


Figure 5.15: Screenshot of generation 30 of the genetic algorithm for site 2 after performing the glitch.


Figure 5.16: Screenshot of generation 30 of the genetic algorithm for site 3 after performing the glitch.

### 5.2.4 Experiment Conclusions

The resulting individuals were able to perform the glitch in two of the three sites. We found more information about the glitch, apparently when the player character is in crouch state and attacks, if the crouch button is released the player stands up without checking if it can stand up. More so, we discovered information on the frame window for releasing the crouch button.

The results do not imply that the glitch can not be performed on the first site though. We believe that the glitch can be performed the same way than in site 3. The effects of the glitch will be different in site 1, because in site 3 the wall is connected to the roof, which is not true in site 1. By trying to perform the glitch in site 1, we expect that the player character gets propelled upwards until it no longer collides with a wall.

## 5.3 Zabuyaki

Zabuyaki is a cross-platform, open source side-scrolling beat'em up. The game has currently one playable level with two fully functional characters and two unfinished characters to pick. The developers frequently update the Github repository of the game.

This game has no public issues, so we contacted the developers in order to ask if they needed assistance with hard to reproduce bugs. Talking with the developers, two issues were brought to light: the D545 task and the D694 task.

The D545 task is about the fighting moves of one of the finished characters named Chai, specifically their special and charge dash attack moves. Both moves have two parts, the

second part is supposed to trigger when the first part lands. These moves present two problems (that will be referred as *points* from now on): the first point is about an infrequent issue that consists of the second part of these moves triggering despite the first part not landing; the other point is about some occasions when the second part of a move is not triggered even after the first part lands.

Now the moves in question will be described in more detail. The special dash move can be performed by dashing (pressing an horizontal directional key two times) and press the attack and jump button at the same time while doing it. The character is supposed do a flying kick first and then perform another kick if the first one lands (Figures 6.11 and 6.12).

The charged dash move on the other hand can be performed by charging (holding the attack button for some time) and achieving maximum charge; then performing a dash and releasing the attack button. The character is supposed to perform a kick in mid-air, and if it hits, the character performs a second kick (Figures 6.13 and 6.14).

The D694 Task is about combining moves to form combos. In some cases the last move of these combos does not hit the target when it looks like it should. The following combos (with the respective characters that performs them) are examples where this phenomenon is said to occur:

- Rick: combo1 → combo2 → dashAttack
- Chai: combo1 → combo2 → dashAttack
- Rick: combo1 → forwardCombo2 → forwardCombo3 → offensiveSpecial

The development team thinks that all these issues are connected and have videos of the issue happening where moves are performed but do not hit despite the hitboxes overlapping. Some information the team finds useful to determine the cause of this issue is the frame windows the last move should be activated for this to happen; and the distance between a player and the target. For the next combos, the frame window on which a move's hitboxes are active will be referred as the length of the move.

The first combo mentioned will be referred to as Rick's Dash Attack Combo. This combo can be performed by attacking two times while not moving, and rapidly performing a dash and pressing the attack button again. The first two moves are punches that last one frame and the final move corresponds to an attack that lasts 20 frames (with two hitboxes that appear in the 11, and 16-20 frame windows), and propels the player character forward in the direction it is facing.

The second combo will be referred to as Chai's Dash Attack Combo and it can be performed in the same way as Rick's Dash Attack Combo. The first two moves are punches that also last one frame, the final move lasts 26 frames and also propels the player character forward in the direction it is facing. This last move has three hitboxes: the first one appears on frames 5 and 6; the second one is active between frames 7 and 10 (including them); and the last hitbox is active between frames 11 and 26.

The final combo will be referred to as Rick's offensive special combo. This combo is performed by pressing the attack button once, then holding one of the horizontal directional

keys and pressing the attack button twice, and finally pressing the attack and jump buttons at the same time. The first move lasts one frame, the second and third lasts 6 and 7 frames respectively and the final move lasts 12 frames and has an active hitbox during frames 11 and 12.

### 5.3.1 D545 Task Point 1

This point refers to when the first part of the moves described before does not hit but the latter is triggered anyways. We first tested the special dash, this move consists on a flying kick, if it hits, it hurts the target six times and then it performs the second part of the move; which is a kick that hits the target one more time. No more detail was provided in the issue, so two fitness functions were designed to reproduce it.

The first fitness function will be called the **No-Hit fitness function**. This fitness function rewards an individual when the game does not detect a hit on the first part but still performs the second part of the move. This fitness function also rewards the player in a lower scale for performing actions similar to the move, so if it does not perform the move on generation 1, the algorithm can increase its fitness by doing similar things that are easier to perform. We defined the following variables to create this fitness function:

- The number of dash attacks performed (d).
- The number of special dash attacks performed ($d_s$).
- The number of special dash attacks that hit ($d_{sl}$).
- The number of times the second part of the special dash attack was triggered ($d_{sl2}$).

As mentioned before, performing similar, easier behavior will also reward the genetic algorithm. In this case, dash attacks will be used as they can be performed in the same way as the target move, except they do not require the use of the jump button. The function is created so it awards increasingly more fitness to an individual the closer it gets to performing the issue.

$$fitness = \frac{d}{8} + d_s + 10\, d_{sl2}(1 + 9\,(d_{sl2} - d_{sl}))$$

The second fitness function will be called **No-Damage fitness function**. This fitness function rewards the player as less damage is produced by the first part of the special dash move. It shares many variables with the first fitness function but it adds one new variable: number of attacks of the move that did 0 damage ($h_0$).

$$fitness = \frac{d}{8} + d_s + 10\, d_{sl}(1 + h_0)$$

For the charged dash, the fitness function developed used new variables as it corresponds to a different move. As the move requires the player to *charge*, the function was designed so individuals would be rewarded for achieving *max charge* in the beginning. We defined the following parameters for developing this function:

- A number between 0 and 1 that represents the maximum amount of charge achieved ($c$).
- The number of charged dash attacks performed ($d_s$).

- The number of charged dash attacks that hit ($d_{sl}$).
- The number of frames the hitboxes of the second part of the charged dash attack were active ($d_{sl2}$). There is a total of 14 of these frames in one complete instance of the move.

$d_{sl2}$ is the variable used to see if the player character performed the second part of the charged dash attack in this function. No other way to detect this using game logic was found, so in order to do it, the fitness function divides this value by 14 to see how many times the second part of the move was performed. Using these parameters the function (which will be called **Charged No-Hit Fitness Function**) is defined as follows:

$$fitness = c + 10\,d_s + 100\,\frac{d_{sl2}}{14}(1 + 9\,(\frac{d_{sl2}}{14} - d_{sl}))$$

If the genetic algorithm does not manage to perform the move in the first generation, it will still be awarded for its charge achieved. The max charge was used instead of frames spent charging to give more exploration to the algorithm once it achieved to max charge. After that, the algorithm would be rewarded for performing the move and the issue similarly as before.

### Special Dash Experiments

For all experiments we manually created an input sequence to skip the menus when evaluating individuals. The individual type chosen was Multi Ephemeral Key Individual; and the individual's and genetic algorithm's parameters were set as follows:

- **Generations:** 20
- **Individuals by Generation:** 100
- **Mutation Probability:** 0.2
- **Elitism Ratio:** 0.05
- **Frames to Skip:** 350
- **Frames to Test:** 800
- **Initial amount of activations per input:** 20
- **Ratio of activated vs deactivated input time:** 5%

The *Initial amount of activations per input* are the amount of times every input is activated during the execution for individuals of the first generation. On the other hand, the *Ratio of activated vs deactivated input time* parameter represents the ratio of time the inputs are hold compare to the time the inputs are deactivated during execution. By setting a high value to the first parameter and a low to the second, the individuals of the first generation will have longer and fewer inputs.

These parameters were defined like this so it is easier for the genetic algorithm to produce input sequences that perform the special dash attack. The number of frames was set to 800 so the algorithm's window to produce the move is not to big. Finally the number of generations was set to 20 in case the algorithm converges too quickly; and to observe preliminary results as soon as possible.

In both experiments, the genetic algorithm was able to perform the special dash attack on enemies but the issue was not replicated. In Figure 5.17 the genetic algorithm's fitness seemed to converge rather quickly as no optimizations were made in the latter generations

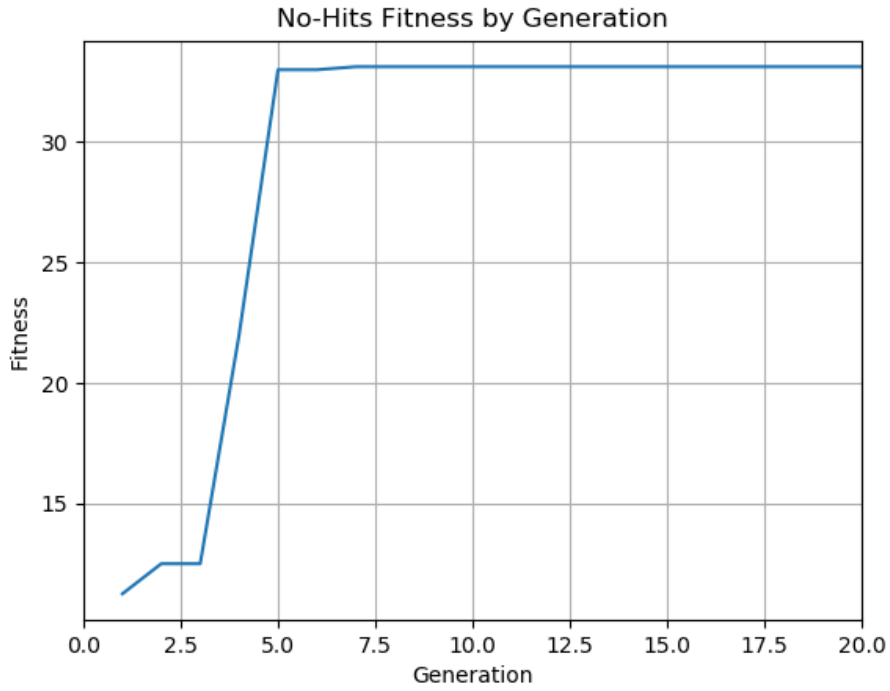


Figure 5.17: Fitness by generation of the genetic algorithm with the No-Hit fitness function.

In Figure 5.18 the genetic algorithm's fitness also converges rapidly with no optimizations observed after generation 10. The best individuals of the last 10 generations all made a kick where just the first hit of the move landed and the rest did not; as the enemy the move was performed on died on the first hit. Thus given these results no more tests were performed with this approach.

### Charged Dash Experiments

The same input sequence to skip the menus as the previous experiment was used in these experiments. For these experiments we used Multi Ephemeral Key Individuals, as simultaneous inputs are required in order to perform a charged dash. The individual's and genetic algorithm's parameters were set as follows:

- **Generations:** 20
- **Individuals by Generation:** 100
- **Mutation Probability:** 0.2
- **Elitism Ratio:** 0.05
- **Frames to Skip:** 350
- **Frames to Test:** 400
- **Initial amount of activations per input:** 4

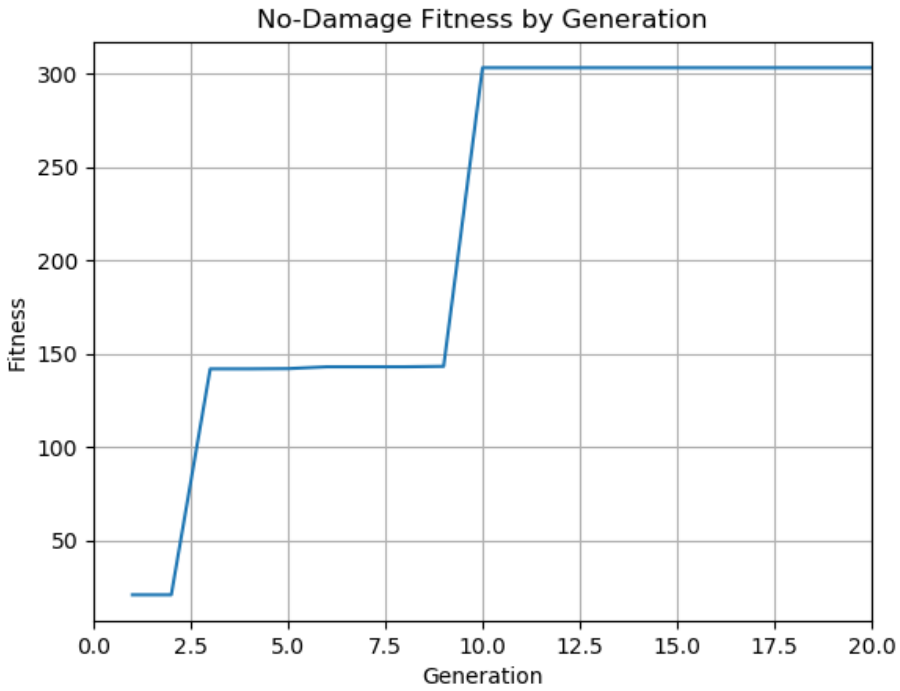


Figure 5.18: Fitness by generation of the genetic algorithm with the No-Damage fitness function.

- **Ratio of activated vs deactivated input time:** 90%

These activation related parameters were set like this so it is easier for the genetic algorithm to get to max charge in order to perform the charged dash. The inputs that the genetic algorithm could do were restricted to the movement inputs and the attack inputs, as other inputs are not necessary for performing the issue. We reduced the frames to test to 400 because of time issues.

Looking at Figure 5.19, it took around 4 generations for the genetic algorithm to produce an individual that could get to a fully charged state. In generation 9, the GA started producing individuals that could land a charged dash. However no more optimizations were observed during the rest of the execution and the issue was not reproduced.

### Experiment Conclusions

For the special dash, both approaches were not effective to reproduce the problem. Given the nature of the approaches used, if this variant of the issue still existed in the game it may be a visual bug where the first attack does not seem to hit but the game detects as it did. Another possibility is that the issue has been fixed since the last time it was reported in August 2018, since the results obtained fit with the ones that would occur if the bug has been fixed.

For the charged dash, it is believed after some analysis that the fitness function designed for the experiment was not completely adequate. This is because of the way that it was

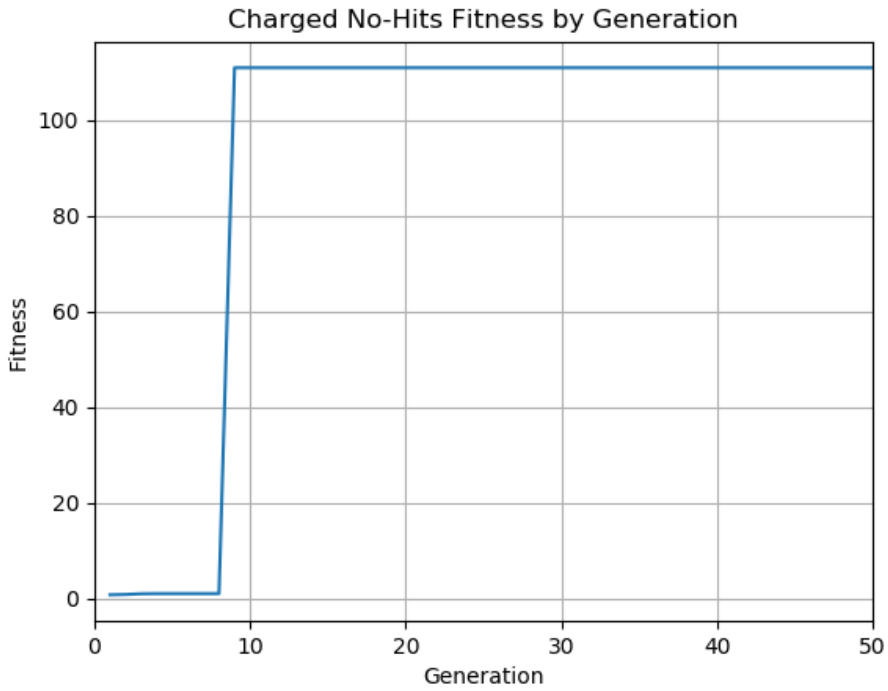


Figure 5.19: A graph that shows the fitness by generation of the genetic algorithm with the Charged No-Hit fitness function.

interpreted that "the first part does not hit but the second part is triggered anyways". It may be possible that by "hit" the developers meant the first part did not do any damage instead of not hitting. However, as the last time the issue was detected was in 2018 no more testing was done in regard to this case of the issue.

### 5.3.2 D545 Task Point 2

This point refers to when the first part of the moves described before lands but the latter is not triggered. For the special dash only one fitness function was tested. This function was designed using the parameters used in the fitness functions of the previous experiment:

$$fitness = \frac{\mathrm{d}}{8} + \mathrm{d}_s + 10\,\mathrm{d}_{sl}(1 + 9\,(\mathrm{d}_{sl1} - \mathrm{d}_{s2}))$$

This function will be called **No-Second Fitness Function**. It is very similar to the **No-Hit Fitness Function** but it rewards an individual greatly for landing the first part of a special dash and avoiding the second part.

For the charged dash, the fitness function designed will be called **Charge-No-Second Fitness Function** and it is defined in a similar manner as the previous functions. It first rewards the individuals to get maximum charge, a prerequisite for performing the charge dash. Then it rewards the individuals for performing the move; and greatly rewards them for performing it without triggering the second part of the move.

$$fitness = c + 10\,\mathrm{d}_s + 100\,\mathrm{d}_{sl1}(1 + 9\,(\mathrm{d}_{sl1} - \mathrm{d}_{s2}))$$

### Special Dash Experiment

We used the same input sequence for skipping the menus for this experiment. Again, the individual type chosen was the Multi Ephemeral Key Individual. We set the individual's and genetic algorithm's parameters to the same values as in the previous experiments:

- **Generations:** 20
- **Individuals by Generation:** 100
- **Mutation Probability:** 0.2
- **Elitism Ratio:** 0.05
- **Frames to Skip:** 350
- **Frames to Test:** 800
- **Initial amount of activations per input:** 20
- **Ratio of activated vs deactivated input time:** 5%

This is done because the objective involves performing the same move performed before. As the individuals produced by the previous experiments were able to do the special dash, the same parameters were used. In Figure 5.20 the progression of the genetic algorithm's fitness through the generations is presented.

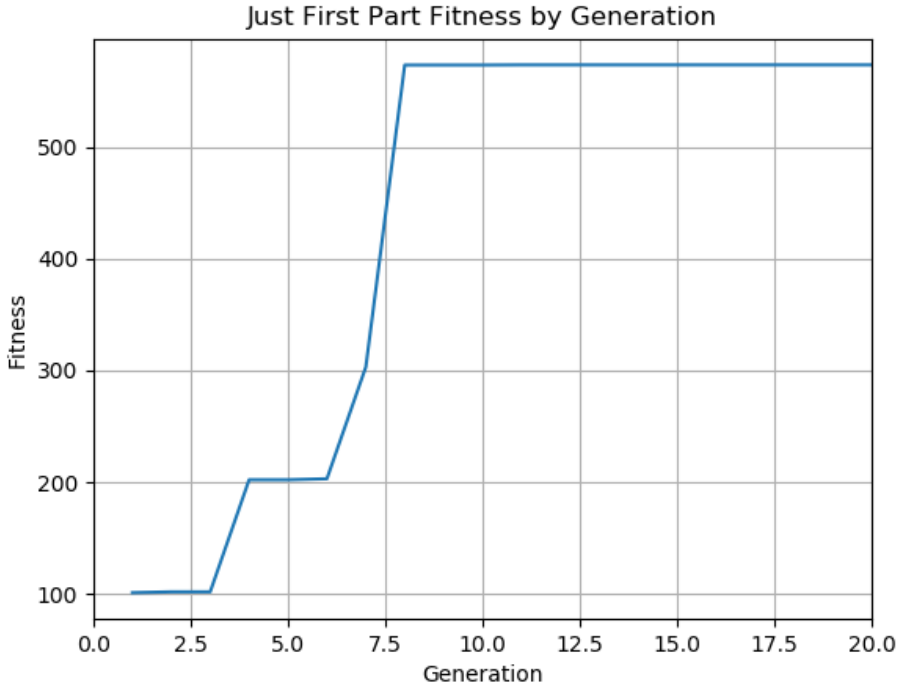


Figure 5.20: Fitness by generation of the genetic algorithm with the No-Second fitness function.

In this case the genetic algorithm managed to reproduce the objective issue by generation 8. When reproducing the issue the player character touched the ground while performing the first kick of the special dash. This caused the player character to fall to the ground, and after that, the rest of the move was not performed.

We did further study on the input sequence the genetic algorithm produced, specifically regarding the z coordinates of the player character during execution. In Figure 5.21, a graph of the altitude of the player character during a regular special dash is shown. In addition, Figure 5.22 shows a graph of the altitude of the player character during the special dash performed by the genetic algorithm.

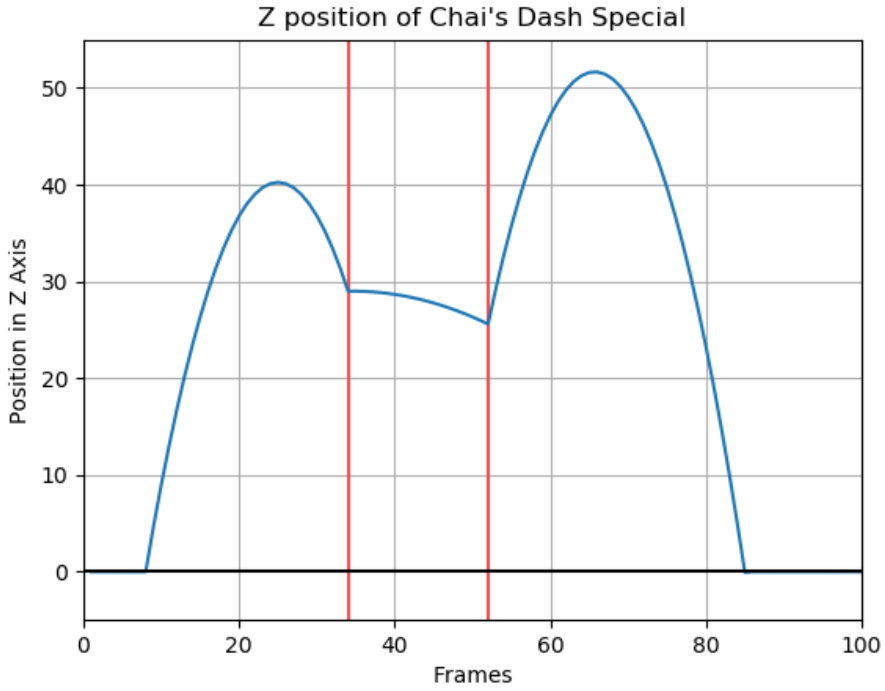


Figure 5.21: Z position of the player character during a special dash that hits an enemy. The first red vertical line shows when the move hits a target. The second line shows when the second kick of the move is performed.

It can be observed in Figure 5.22, that after landing the first kick, the z position of the player character stayed at zero. Looking at Figure 5.23, it seems that around 3 frames after the first kick landed the player character made contact with the ground. After this, the player character's altitude was reset to the ground level and the rest of the move was cancelled.

### Charged Dash Experiment

For this experiment we also used Multi Ephemeral Key Individuals and the same input sequence to skip menus as the previous experiments. The parameters for this experiment were defined as follows:

- **Generations:** 50
- **Individuals by Generation:** 100
- **Mutation Probability:** 0.2
- **Elitism Ratio:** 0.05
- **Frames to Skip:** 350

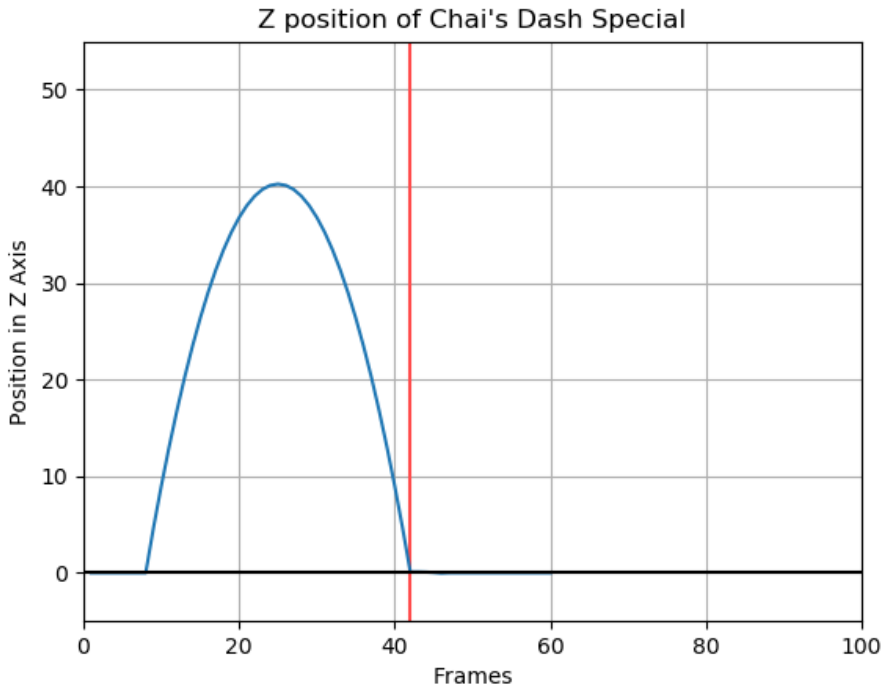


Figure 5.22: Z position of the player character during a special dash performed by the genetic algorithm. The red vertical line shows when the move hits a target.

- **Frames to Test:** 400
- **Initial amount of activations per input:** 4
- **Ratio of activated vs deactivated input time:** 90%

The amount of frames to test was reduced to 400 because of time issues, but the issue can be performed in a much smaller amount of frames than that, so it was not seen as a problem. The ratio of activated vs deactivated input time was turned up to 90% so hold inputs are more frequent, and so it becomes easier for the individuals to "charge". For this same reason reducing drastically the initial amount of activations per input would mean that less inputs are being activated and so, inputs are being hold for more time.

In this experiment, because of the way some data is obtained, the genetic algorithm can cheat by performing the first part of the move just at the end of the simulation. Because of this, the amount of frames to test given to the wrapper are 500 but the input sequences tested will just last 400 frames. This way if an individual performs the move at the end of execution without performing the issue, the fitness function will not reward this individual.

From Figure 5.24 it can be seen that during the first 15 generations, the individuals were learning to charge. In generation 16 they started to perform the move, finally in generation 19 the individuals landed the move on an enemy. No more improvements were made by the genetic algorithm in the next generations, getting a higher fitness than the one obtained here would have meant performing the issue. Using the data from resulting individuals, a similar analysis as the special dash was done with the charged dash.

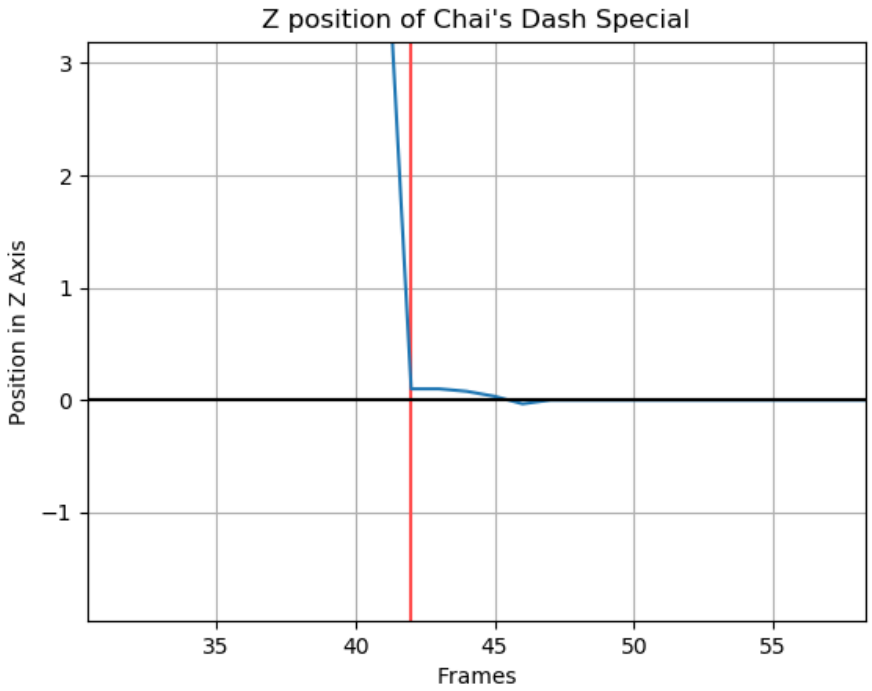


Figure 5.23: Z position of the player character during a special dash performed by the genetic algorithm. The red vertical line shows when the move hits a target. This graph was zoomed around the moment the first kick of the move landed.

The z position was recorded while the individual performed the charged dash and the results can be seen in Figure 5.25. It can be observed that the z position when the move hits is lower than that of the last frame of the active hitboxes from the second part of the move. This means it is not possible for the individual to touch the ground while these hitboxes are active.

#### Experiment Conclusions

For the special dash, the issue seems to be caused because the player character can touch the ground during the first part of the move. The developers believe that this is the reason the rest of the move gets canceled. This happens during the time the first kick is doing damage as the z position of the player decreases during this interval since it lands.

For the charged dash, the same cause of the issue does not seem to be possible as all damage is done in a position higher than the position when the move hits. This does not mean the issue is not present in the game anymore, as the definition of task itself is somewhat ambiguous. However as no instance of the issue happening have been reported since August 2018, no further experimentation was done.

### 5.3.3 D694 Task

We attempted to reproduce two of the three combos mentioned before with the GA. This is because the issue was reproduced during the experiments for the second combo, so it was

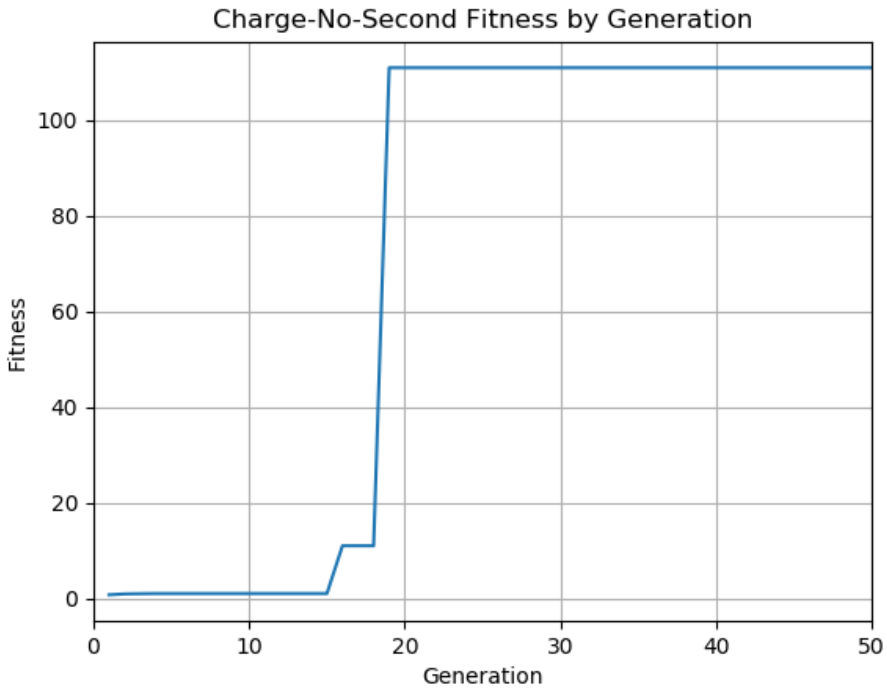


Figure 5.24: The fitness by generation for the charge no second fitness by generation.

decided that doing the experiments with the last combo was unnecessary. The combos we tested were: Rick's Dash Attack Combo and Chai's Dash Attack Combo.

Reproducing combos is a complex task, mainly because of how to obtain information. It is easy to determine through the use of game logic when a specific move is performed, but the game does not have internal parameters that indicate if a combo is being performed. For these experiments, we define a combo as the act of chaining two or more moves rapidly, landing every one of them on one target. Thus a strict heuristic must be developed to determine when a combo is being performed.

The heuristic we designed for the experiments is simple. We assume that if a move is triggered 10 frames after the previous move (that must have landed) is triggered, then a combo is being performed. This heuristic is not very accurate as some moves may take more than 10 frames to perform (as shown in the D545 Task experiments), but it will suffice for the three combos to reproduce.

Going back to the objective of the task, reproducing this issue with genetic algorithms proved to be very hard. This is because there are many ways to perform the objective behavior in a way it is not the issue. Taking Rick's Dash Attack Combo for example, the second move can kill the target and then the dash attack can be performed; the second move may hit an enemy in mid air, which leaves them out of the reach of the dash attack; the player character can turn around to perform the dash attack thus not hitting the target in front of them; etc.

Because of the many possibilities this behavior has, some restrictions were applied to the

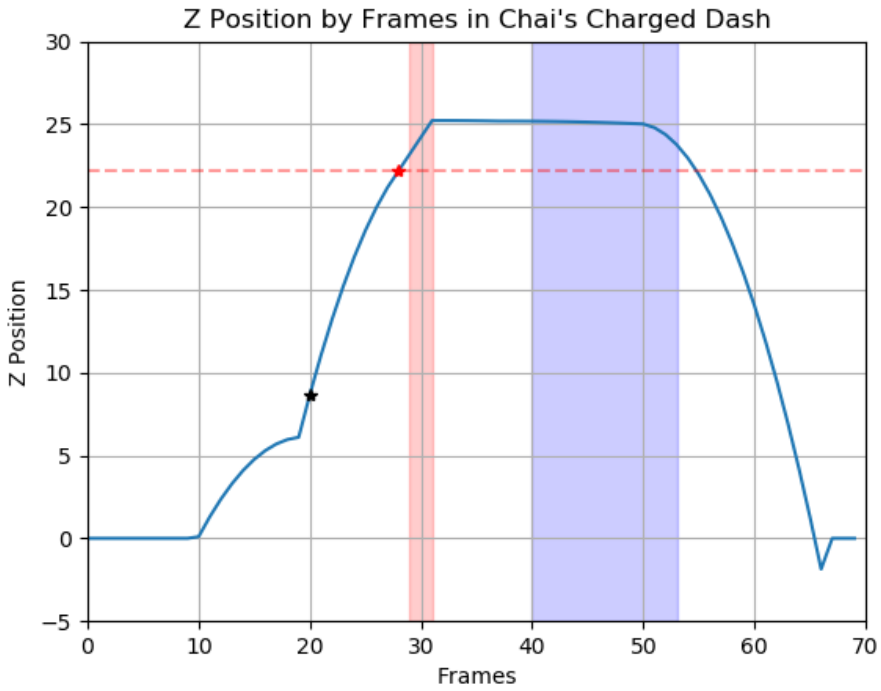


Figure 5.25: A graph that represents the z position of Chai during a charged dash. The black star shows the frame the game detected when the charged dash was triggered. The red star shows the frame the charged dash landed. Finally the red area represents the frames where the hitboxes of the first part of the move were active; and the blue area is the same for the second part of the move.

genetic algorithm and the fitness function. First, the individuals generated by the genetic algorithm can only move up, down and right, also they can not press the jump button. Second, every move except for the last one must do at least 1 damage in order to be considered the combo to be reproduced. Finally, killing an enemy will result in a fitness of 0.

Thus the following parameters were used to create the fitness function for Rick's Dash Attack Combo:

- $c_1$, a variable that equals 1 if Rick's combo1 was performed and 0 otherwise.
- $c_2$, a variable that equals 1 if Rick's combo2 was performed and made damage, and 0 otherwise. For this combo to be triggered combo1 must have hit before so it is not necessary to check if combo1 hit.
- $c_{da}$, a variable that equals 1 if Rick's Dash Attack Combo is performed and 0 otherwise.
- $d_{da}$, a variable that equals the minimum amount of damage a dash attack at the end of Rick's Dash Attack Combo has made.
- $k$, a variable that equals 0 if at least one enemy is killed and 1 otherwise.

$$fitness = k\,(0.1\,c_1 + c_2 + \tfrac{c_{da}}{1+d_{da}})$$

The fitness function developed with these parameters (called Rick's Dash Attack Combo

Fitness Function) rewards individuals at the start for performing the first part of the combo. After that, the individuals that perform the combo by doing the minimum amount of damage with the last move are awarded more fitness. Given the fitness function, an individual that obtains a fitness of 101.1 has successfully performed the issue.

### Rick's Dash Attack Combo Experiments

This was the first combo tested by the genetic algorithm. We created a new skip input sequence to for these experiments; this sequence selects Rick as the player character and activates some dev tools for visualizing entity hitboxes. The type of individuals used was Multi Ephemeral Key Individual and the parameters used for the experiments are:

- **Generations:** 20-50
- **Individuals by Generation:** 100
- **Mutation Probability:** 0.2
- **Elitism Ratio:** 0.05
- **Frames to Skip:** 450
- **Frames to Test:** 400
- **Initial amount of activations per input:** 10
- **Ratio of activated vs deactivated input time:** 10%

In total, 20 experiments were done for this instance of the issue, the results of each experiment were used to iterate over the fitness function.

The number of generations for the experiments started in 50 and was then reduced to 20 because the genetic algorithm tended to maximize the fitness function in less than 15 generations. Many experiments were necessary in order to adjust the fitness function so the genetic algorithm could reproduce the objective behavior. In the first 8 attempts, the genetic algorithm managed to reproduce the behavior described without performing the issue.

In following attempts the accepted frames between moves for the combo was increased to 15 and later to 20. For the experiments with 15 frames, the genetic algorithm managed to produce individuals that only hit the first hitbox of the dash attack. In the experiments with 20 frames between moves, the genetic algorithm managed to manipulate the AI of the game to get out of the way between moves, obtaining maximum fitness.

In the final attempts, the amount of frames was reduced and another player character was summoned to serve as a punching bag. However, no improvements were observed, the genetic algorithm did not seemed to be closer to reproducing the issue with these approaches. Because of this, no more experimentation was done with Rick's Dash Attack Combo.

The resulting fitness function from these experiments seemed to be able to guide the genetic algorithm to perform the objective combo. Thus this same function was adapted for the following experiments.

### Chai's Dash Attack Combo Experiments

For this combo, we developed a variation of Rick's Dash Attack Combo's skip input sequence, but instead of choosing Rick, it chooses Chai as the player character. We used Multi Ephemeral Key individuals for the experiment, and the parameters used for the genetic algorithm are:

- **Generations:** 50
- **Individuals by Generation:** 100
- **Mutation Probability:** 0.2
- **Elitism Ratio:** 0.05
- **Frames to Skip:** 400
- **Frames to Test:** 500
- **Initial amount of activations per input:** 10
- **Ratio of activated vs deactivated input time:** 10%

For this experiment, the accepted frames between moves for a combo tested were 15 and 20. The individuals with the frame window of 15 were able to reproduce the objective combo, however they were not able to reproduce the issue. In Figure 5.26 the fitness by generation for this experiment can be observed.

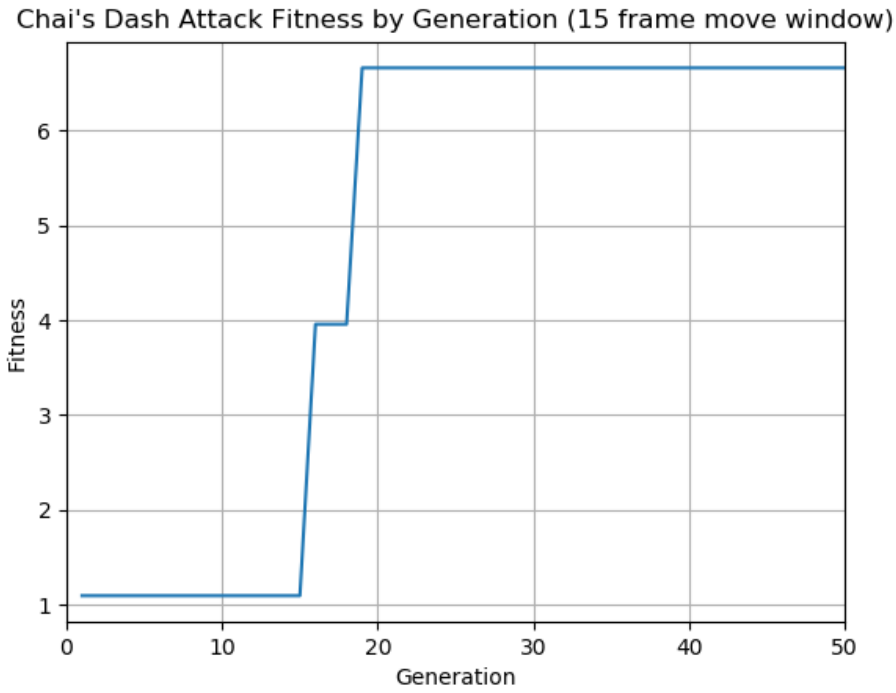


Figure 5.26: A graph of Chai's Dash Attack fitness by generation with a 15 move frame window.

In the experiment from Figure 5.26, the first 15 generations the individuals managed to just get the first two moves of the combo. Between generations 16 and 18 the individuals were

combo1 → combo2 → combo3Forward → dashAttack

Figure 5.27: Combo performed by the genetic algorithm using the player character Chai that managed to reproduce the D694 Task.

able to do the objective combo hitting enemies in two different frames. From generations 19 and onwards the individuals managed to do the objective combo hitting enemies in just one of the frames.

The individuals with the frame window of 20 were able to reproduce the issue mentioned in the D694 Task. The combo they performed was not Chai's Dash Attack Combo, instead they did the combo shown in Figure 5.27. A graph with this last run of the genetic algorithm's fitness through generations can be observed in Figure 5.28.

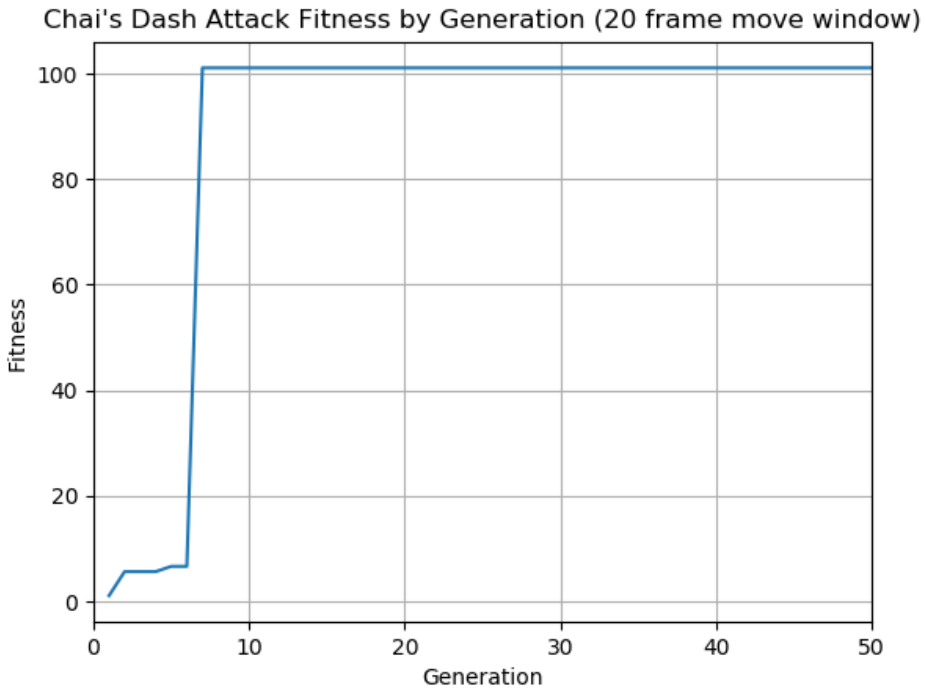


Figure 5.28: A graph of Chai's Dash Dash Attack fitness by generation with a 20 move frame window.

In the first generation of this execution, the genetic algorithm produced an individual that performs the first two moves of the combo. In generation 2 the GA managed to perform Chai's Dash Attack Combo on an enemy. In generation 5 and 6 the fitness increased but no noticeable difference was observed. From generation 7 onward the individuals produced by the genetic algorithm performed the issue with the new combo, which we named Chai's Forward Combo.

We contacted the developers because of the reproduction of this issue and they requested an analysis of the timing of the move and the distance on which the issue is produced. Thus we did an analysis of the timing of the move by modifying the frame the input sequence performed the dash attack. The results of this analysis are as follows:

- **Frames 2-3, 13 and 16:** The combo3Forward does damage but the dash attack is not performed.
- **Frame 4:** The combo3Forward gets cancelled mid animation before it can do damage and the issue is performed.

  **Frames 5-12:** The combo3Forward does damage and the issue is performed, the dash attack does not do any damage.

  **Frames 14-15 and 17:** The combo3Forward does damage and the dash attack is performed and does damage as normal.

  **Frames 18-20:** The combo3Forward does damage and combo4Forward is performed instead of the dash attack.

A distance analysis could not be done using the same input sequence as before as it performed the issue on AI controlled enemies. Changing the distance by editing the existing input sequence would change the AI behavior, so the target of the combo could not be an AI. When speaking with the developers they provided some instructions to use some debug tools to get rid of enemy controlled players and summon another player character to use a training dummy. Following these instructions, we developed a new skip input sequence and the genetic algorithm was ran the same as before with some small exceptions.

As now there were no enemies that can make the controlled player character turn around, the restriction that the player can not turn was removed. The restriction that no enemy can die was also removed as a lot of enemies died because of the debug tools used. Having these two restrictions removed, the genetic algorithm's parameters were set the same as in the previous experiment and a new input sequence that reproduces the issue was produced.

We produced the new input sequence, the initial horizontal distance difference between the player character and their target was manually varied by having the other player character move horizontally. The initial distance between the player characters when the first move of the combo hit, was recorded for many distances. After that the behavior of the game was recorded for each distance and grouped by similar behavior. The results of the analysis are as follows:

- **Distance 0 - 11 units:** The third move of the combo (combo3Forward) does not hit and the dash attack works as normal.
- **Distance 12.6 - 47.6 units:** The first three moves of the combo hit and the issue occurs, the dash attack does no damage.
- **Distance 49.3 - 59.3 units:** The first two moves of the combo do not hit; Chai performs a combo2Forward that hits because of this. The issue occurs, the dash attack does no damage.
- **Distance 61 units - Onwards:** The first three moves of the combo do not hit. The dash attack is performed as normal.

### Experiment Conclusions

As shown by Rick's Dash Attack experiments a lot of experimentation was needed in order to get the fitness function to get close to the target behavior. This task consisted on complex behavior as a lot of factors had to be taken into account. Between these factors are that the

player character should not be able to turn around during a combo; no enemies should die because otherwise the genetic algorithm could cheat by killing an enemy with the second last move of the combo; etc.

Despite the results of Rick's Dash Attack experiments, the fitness function developed was able to make the genetic algorithm perform the target combo. Thus the fitness function designed for Chai's Dash Attack experiments was only an adaptation of this fitness function. And as shown in the results of the experiment, this fitness function was able to reproduce the issue for this move. Given these results, it is believed that it is easier to perform the issue with Chai's Forward Combo, and that is the reason why the function worked for this experiment but not the previous one.

The analyses of the issue performed over input sequences generated by performing Chai's Forward Combo revealed important information about the issue. The timing analysis showed that varying the timing of the dash attack, the issue was performed in frames 5 to 12 and that the combo3Forward always does damage when the issue is performed. The distance analysis on the other hand, further expands on this claim as the instances when the issue occurs here are when the combo3Forward or combo2Forward are performed and hit the target before performing the dash attack.

## 5.4 Time Analysis

As mentioned in the problem section, it is important that the solution developed does not take more than one day in obtaining some meaningful results. Because of this statement, as part of the validation process for this software, some time metrics were computed in order to obtain the approximate time of execution in a general case. These metrics were computed by replicating the experiments done to *Journey to the Center of Hawkthorn* and *Zabuyaki*.

The metrics computed correspond to the average testing time by individual and the average crossover time. These metrics were obtained in experiments to understand how do the software's execution time varies for different amount of frames tested. Some of these experiments were defined in order to test some features of the software that accelerates the execution.

The individual type used for these experiments were the Multi Ephemeral Key Individual. The genetic algorithm's parameters were defined as follows:

- **Generations:** 30
- **Individuals by Generation:** 100
- **Mutation Probability:** 0.2
- **Elitism Ratio:** 0.05
- **Initial amount of activations per input:** 10
- **Ratio of activated vs deactivated input time:** 10%

The first experiment's objective was to measure the execution time to test individuals simultaneously using various amount of threads. The next experiment was made in order to compare frames tested and execution time for Zabuyaki. The next experiment compares

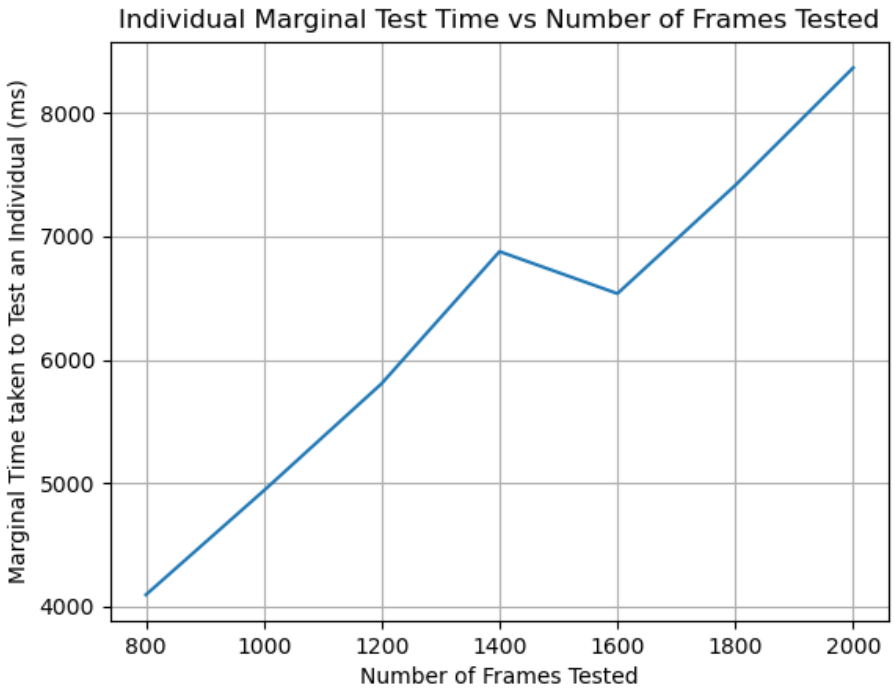


Figure 5.29: A visualization of the relationship between the amount of frames tested and the time taken to test a single individual on average for Journey to the Center of Hawkthorn.

execution time and the frame interval, and was done over Journey to the Center of Hawkthorn. The last experiment compares the frames tested and the execution time for Hawkthorn.

When doing this last experiments, the hardware used had to be reset due to a memory problem. Because of this, the results of the last experiment present a particularity. However due to time restrictions it was not possible to repeat them.

### 5.4.1 Testing Time Analysis

For the testing time analysis, we designed an experiment for both games. For Hawkthorn, we used the custom testers designed so no input sequence to skip frames had to be developed. For Zabuyaki, we used the skip input sequence from the D694 Chai's Dash Attack Combo experiments.

In both cases the total length (frames to test plus frames to skip) of the input sequences tested ranged from 800 to 2000 in intervals of 200 frames. For each generation, we calculated the total time used for fitness computation and reproduction. In this section, the fitness computation time was used in order get an approximate time it takes to test a single individual. It is worth mentioning that the amount of threads used for these experiments is equal to 5. The results can be seen in Figures 5.29 and 5.30.

It can be observed in Figure 5.29 that the growth is approximately linear from 800 and 1400 frames tested. Then it decreases when testing 1600 frames and continues to grow linearly after that. This reduction in time is what is referred as the particularity of this experiment

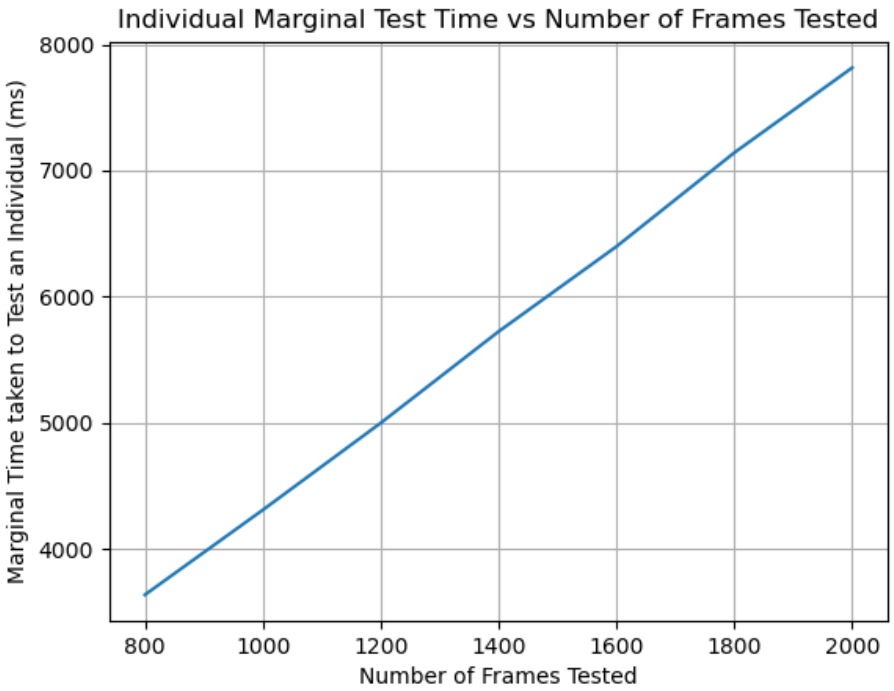


Figure 5.30: A visualization of the relationship between the amount of frames tested and the time taken to test a single individual on average for Zabuyaki.

in the earlier paragraphs.

Despite this, by looking at both Figures the relationship between these variables seems to be linear. Which makes sense if every frame takes the same to run, as doubling the amount of frames should double the total execution time.

### 5.4.2 Reproduction Time Analysis

The data for the reproduction time analysis for both games was obtained during the same experiments as the data from the previous subsection. This means that these results for Hawkthorn's case are subjected to the particularity previously mentioned.

As mentioned in the previous section, no skip input sequence was used for Hawkthorn and for Zabuyaki, a previously used input sequence to skip the menus was used. We used 5 threads for this experiment and a range of frames tested of 800 to 2000 in intervals of 200 frames. The results of these experiments are shown in Figures 5.31 and 5.32.

Looking at Figure 5.32 it is not clear if the number of frames tested and the average reproduction time by individual are related. It can be observed that all the obtained values are between 0.3 and 0.4 ms by individual in average, so this value varies in a small manner during the experiments.

In Figure 5.31 the variation between values is a lot bigger. It can be observed that between 800 and 1400 frames the time taken scales consistently, then it decreases in the next value and

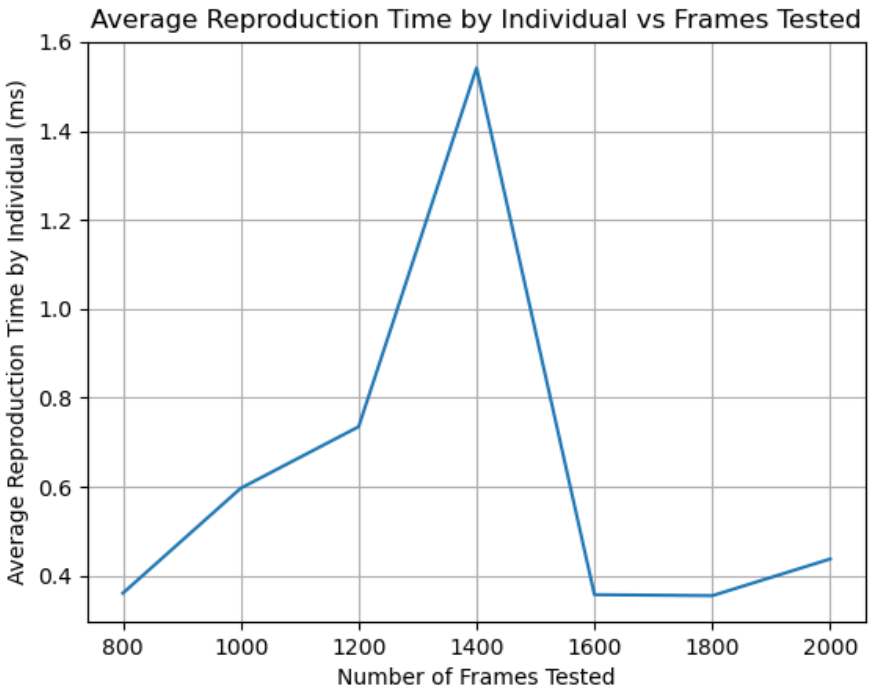


Figure 5.31: A visualization of the relationship between the amount of frames tested and the time taken to create a single individual between generations on average for Journey to the Center of Hawkthorn.

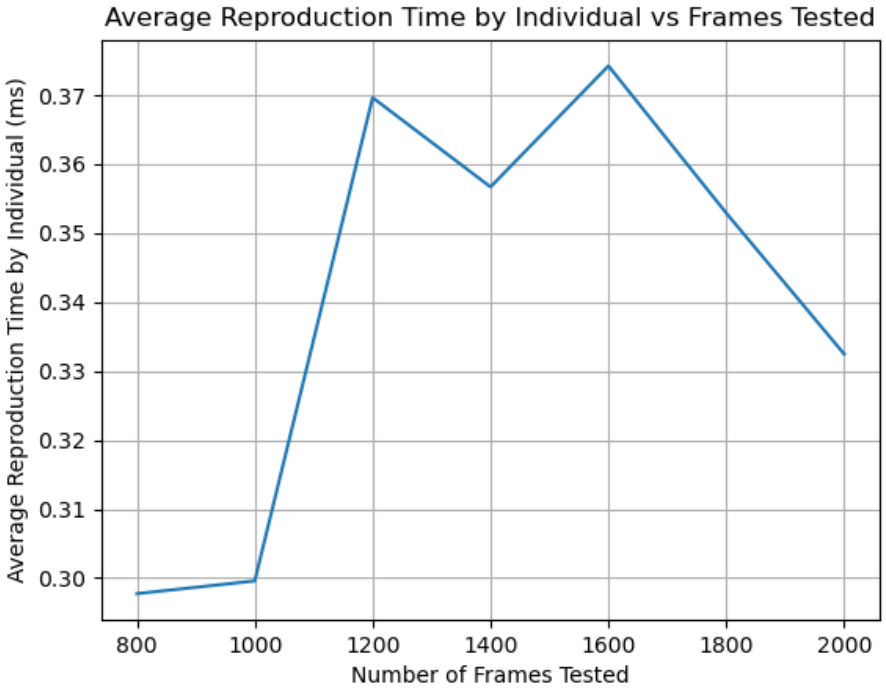


Figure 5.32: A visualization of the relationship between the amount of frames tested and the time taken to create a single individual between generations on average for Zabuyaki.

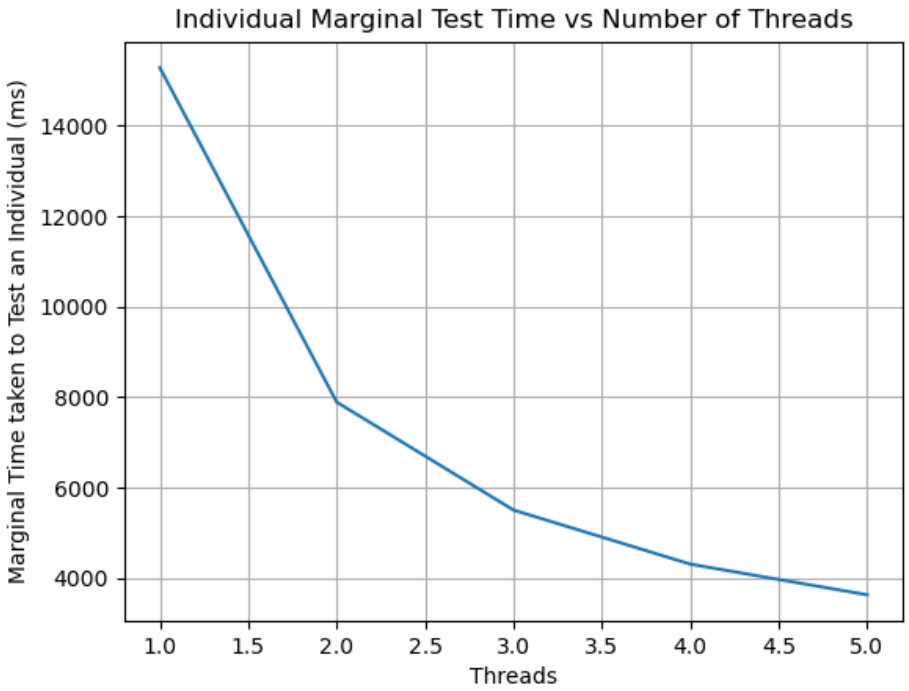


Figure 5.33: A visualization of the relationship between the amount of threads used to test and the time taken to test a single individual on average for Zabuyaki.

starts to scale again. We believe that the particularity mentioned before is the responsible of this behavior. As every experiment before was done in a consecutive manner but the time elapsed started to scale in this experiments, it may be possible that this last game presents a memory leak that slows the testing process of consecutive experiments.

### 5.4.3 Time Acceleration Feature Analysis

The first one of this type of experiments is the "threads" experiment. The game tested for this experiment was Zabuyaki, so the same skip input sequence mentioned before is used for skipping menus and selecting Chai. The amount of frames tested throughout the experiment was defined as 800 and the amount of threads tested ranged from 1 to 5 with intervals of one thread.

The results of this experiment can be observed in Figure 5.32. It is easy to observe the positive results of this feature, as the amount of time taken to test an individual seems to be approximately inversely proportional to the amount of threads used during testing. Achieving a speedup of around 3 and 4 times by comparing the use of one thread versus five.

The frame interval parameter of the genetic algorithm was also tested in order to see the possible speedup. This parameter was set to 1 during the previous experiments in the validation section. It is incompatible with Zabuyaki because of some hardcoded variables, but it is compatible with Journey with the Center of Hawkthorn.

The frame interval parameter is the interval of frames that advance between each time

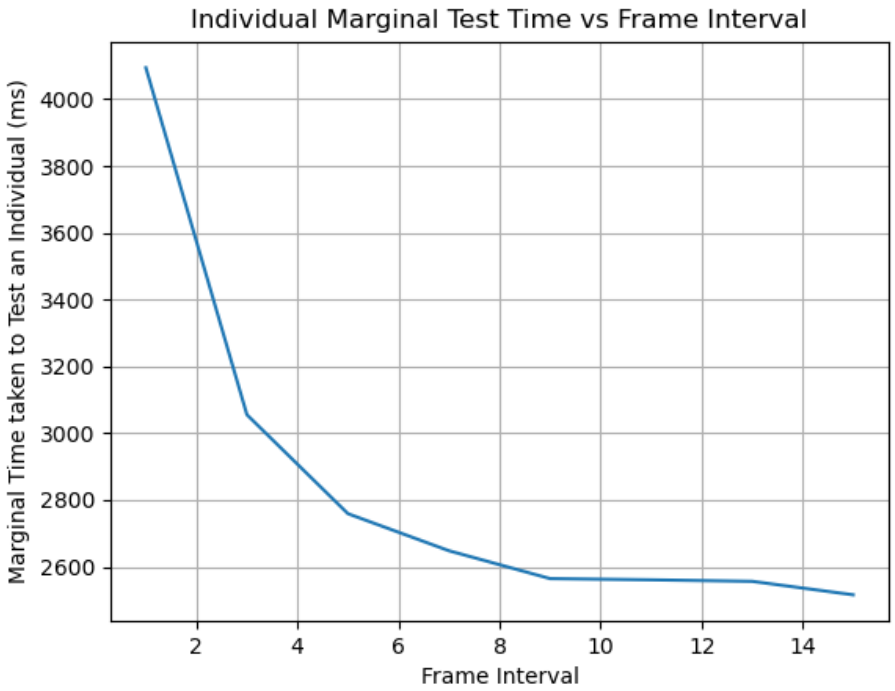


Figure 5.34: A visualization of the relationship between the value of the frame interval parameter and the time taken to test a single individual on average for Zabuyaki.

a frame is elapsed in the LÖVE framework. This process calls the draw function and the garbage collector, but as the update function is called independently by the wrapper it does not intervene with the game logic. It is logical that drawing less frames in the screen would show some kind of speedup in the total execution time.

We set the amount of threads for this experiment to 5. The amount of frames to test was set to 800 and the same parameters, testers, and other as the Hawkthorn time experiments were used. The results can be observed in Figure 5.34.

It can be observed in the Figure that increasing the frame interval parameter indeed accelerates the program. The total speed obtained by increasing its value from 1 to 15 is around 1.5. Around a value of 9 it can be seen that no much decrease in time is observed. It is believed that this occurs because the LÖVE framework has a maximum amount of frames that can elapse each second. And with a frame interval of 9, the maximum amount of frames that are processed each second is close the maximum amount of frames the hardware can process, bypassing the framework's limitations.

### 5.4.4 Experiment Conclusions

According to the experiments done, there seems to be a linear relationship between the amount of frames tested and the testing execution time, as one would expect. The reproduction time seems to not be related to the amount of frames tested, which makes sense as the selection and crossover algorithms depend on the amount of individuals in the population

and the amount of genes these individuals have.

Regarding the acceleration features' experiments, it can be observed that both features decrease the execution time to test an individual. The frame interval parameter should achieve greater results if a game processes less things every frame. On the other hand the threads feature works independent of the game and every game should have the same speedup shown in the experiment given the same hardware.

Given the results of the experiments it can be observed that it takes approximately less than 5 seconds to test a single individual given the conditions used for the previous Hawkthorn and Zabuyaki experiments and 5 threads to test. Taking into account that the reproduction process by individual takes around 0.4 milliseconds, and assuming the initialization of each individual takes a similar amount of time we can estimate how much the Zabuyaki and Hawkthorn experiments took to execute. This value would be given by the following formula:

$$t(g, \mathrm{i}) \approx g\mathrm{i}(5 + 0.4)$$

Where $g$ is the number of generations tested and i is the amount of individuals in the population. For the Hawkthorn experiments 30 generations were tested with a population of size 100; which put in the formula would give a value of 16200 seconds or 4 hours and 30 minutes. For the Zabuyaki experiments, the longest one had 50 generations instead of 30; changing this parameter it would give us a value of 27000 seconds or 7 hours and 30 minutes.

Finally, because of the particularity of the Hawkthorn experiments it would be hard to say for sure if the rest of the experiments are valid. Given the reproduction time of Zabuyaki and Hawkthorn it would seem that the memory problems started when testing this latter game. In order to get more accurate results the experiments should be redone resetting the hardware to clean the RAM memory between each experiment.

# Chapter 6

# Conclusion

## 6.1 Summary of the Work Done

As a result of the work done during the development of this project, two pieces of software were implemented. A wrapper of the LÖVE framework created to run input sequences over games and to compute metrics and user-defined functions. And a software for generating input sequences using genetic algorithms, which used the wrapper to compute the fitness of individuals.

After implementing a preliminary version of the first software, the game *luaAsteroids* was used to test encodings for the genetic algorithm. The game had a memory problem where it did not de-spawn the asteroids that got out of the screen, so after we were able to reproduce the memory leak with a simple encoding we started experimenting with different encodings. During these experiments the Ephemeral Key Individual was created; and from the implementation of this individual the Multi Input Ephemeral Key Individual was later made.

After these experiments, we tested the game *Journey to the Center of Hawkthorn*; the public issues posted in the Github repository were reviewed and two of them were selected to be reproduced. The first one was about inconsistent physics in moving platforms and the second one was about an out-of-bounds glitch that lets the player get inside walls in certain levels. Both of these issues were reported in 2016 and some general information was known about them, but not specific information like frame windows for the inputs.

Some optimizations were made to the genetic algorithm so it took less time to run an experiment. These included not drawing some frames and parallel individual evaluation for the genetic algorithm. These also included the creation of input sequences to reset the game's external storage (cleaning save files for example) and to skip parts of a game (going through menus and other similar tasks).

We were not able to reproduce the first issue mentioned for this game, and believe it is due to a poor design of the fitness function developed. However the results were used to further improve the mutation function for the individuals used in the genetic algorithm.

The second issue was reproduced in two different levels of the game. A sequence of three inputs was discovered to produce the bug that matches the previously known way of reproducing the bug. The frame windows and timing for every input was discovered by manually modifying the sequence, and the results were posted on the issue in the Github repository.

Finally the last game tested was a side-scrolling beat-em up called Zabuyaki. The issues for this game were obtained by contacting the developers. Two issues were tackled: one about problems with two part moves and one about the last moves of some combos sometimes not hitting the target.

The first issue was reproduced for one of the moves called *Chai's Special Dash*. An analysis of the altitude of the player character was done over the input sequence produced. This analysis was then sent to the developers and helped to confirm a theory they had regarding the issue.

The second issue could be reproduced by the genetic algorithm performing a combo with Chai (Figure 5.27). The developers asked for two types of analyses for this sequence: an analysis of the frame window for the last move for the issue to occur, and an analysis of the distance to the target necessary for the issue to occur. The first analysis was pretty simple as it just consisted of manually modifying the input sequence; this analysis revealed that the issue always occurred when the combo3Forward move hit the target previously.

For the second analysis a new input sequence was required as modifying the position of the input sequence could make the AI of the enemies to change their behavior. This new input sequence was created so the target would be other player character and no other entities would be around. The distance was varied by varying the initial position of the second player character. The results of this analysis also revealed that the issue can be performed with just a combo2Forward followed by a dash attack.

## 6.2 Objective Completion Review

A tool that lets developers test their games using genetic algorithms was successfully developed. The tool is able to run over simple games in LÖVE, and it allows developers to create their own fitness functions in order to produce behaviors in games. It is also created in a modular way, so developers can implement and customize the genetic algorithm's individuals and processes.

A downside is that the tool is complicated to use and needs some tinkering with the game files to work. The tool can only be run through command line instructions. And in order to edit the parameters of the algorithm and the wrapper, three configuration files must be edited. Because the tool is not easy to use, the analysis of the results obtained by the genetic algorithm had to be performed on our end guided by the respective development teams.

The wrapper is not capable of injecting all possible types of inputs that the framework supports. The only type of inputs the wrapper can work on are keyboard inputs. Other inputs work as normal when the wrapper is running over a game.

The wrapper has difficulties running games that overwrite methods from the framework. There is a lot of third party software that does this in order to implement some programming design patterns. An example of this kind of software is the hump.gamestate library[1], a library for using the state pattern in order to handle menus, game scenes and applications.

Most desirable features were not implemented because of time reasons. A desirable feature that was implemented successfully is the metric computation feature. The input sequence reduction feature is implemented but have not been tested. Input sequence visualization and related features were not implemented.

On the experiments side, various issues were reproduced and new information about each issue was obtained by analyzing the resulting input sequences. The developers of the games were contacted and they responded in the case of Zabuyaki. In this last case the information provided helped the developers confirm theories about the issues and further understand them.

## 6.3 Results Analysis

The approach used for applying genetic algorithms to video game testing left very promising results. In every experiment, most complications were due to the complex task of designing a fitness function with low information. When reproducing an issue, the more precise information about the target behavior is available, the easier it is to design a fitness function in order to reproduce it.

In the case of the platform experiments of Journey to the Center of Hawkthorn, no specific cases were reported, the only information available about the issue is that moving wildly may cause the player to glitch inside the platform. Because of this, even though the fitness function used was not well designed given the information available, making a precise fitness function was a very difficult task.

For the second issue in the game, the behavior was easy to reproduce, thus a simple fitness function was enough to perform the issue. This is confirmed by the results of the experiment in the second site for this issue. There the genetic algorithm reproduced the issue on generation 1 as the reproduction was very simple.

For the Zabuyaki experiments more complex fitness functions had to be developed in order to reproduce the issues. Both the D545 and D694 task referred to problems that occurred sometimes when doing a specific behavior. The description of the issues was ambiguous, but the description of the behaviors needed for them to occur was very precise.

For these problems the fitness functions used were designed taking into account that the genetic algorithm may not produce individuals that perform the target behavior right away, thus, easier and similar behaviors were also rewarded in a smaller manner in order to eventually produce individuals that did the complete behavior. These techniques were then polished for the D694 task, leading to the design of fitness functions based on milestones.

[1] `https://hump.readthedocs.io/en/latest/gamestate.html`

The D694 task could be easily divided in steps to achieve in order to get closer to the issue. In this case, the steps were performing each move of the combo in order and finally producing the issue. The fitness function for these last experiments rewarded individuals for achieving the complete combo part by part. First it rewarded 1 point for doing the first move of the combo; then it rewarded 0.1 points for doing the second move **after the first**; finally it rewarded the individual for doing as little damage as possible with the last move of the combo, achieving maximum fitness when no damage is done.

These fitness functions based on milestones should work for every task that can be broken down to steps. But for the reproduction of an issue to be broken down in steps, precise information must be known about the target behavior.

The time experiments done in the validation section confirm the achievement of the quality requirements enunciated in the problem section. The results show the software took less than a day to produce meaningful results in the experiments.

## 6.4 Discussion

There are few works that talk about applying genetic algorithms to video game testing. Most of the related work is oriented to applying genetic algorithms to a specific genre of game with a very clear structure. This is the case of ICARUS [16], an artificial intelligence based on GA created to complete point and click adventure games. Other example of related work that applies GA to video games is Sethbling's mar.io[2], a software that uses the NEAT algorithm to create a neural network that plays Super Mario World [3].

The approach presented in this document can be applied to a video game of any genre and structure, as long as the game is deterministic under the circumstances mentioned in the solution section. This includes a very wide variety of games including: platformers, beat'em ups, racing games, rpg games, etc.

The approach used works well with short input sequences, which may not be true for longer input sequences. This is because when producing longer input sequences, the mutation process may become too random. And so, the probability of individuals increasing their fitness with a random mutation may become too low.

There is a simple way to solve this second issue in order to produce a large input sequence with the approach used. To do this the input sequence could be produced part by part. For example producing an input sequence of 200 frames, then using the result as a skip input sequence and produce the next 200 frames and so on.

This means that compared to having human testers working on these issues, these techniques would work with glitches that can be reproduced in a small amount of frames (500 for example) or with a greedy approach. Additionally, if a glitch is not easy to detect using only game logic, human testers may be better for the task as they can interpret not detailed instructions better. But for tasks that are not hard to detect through game logic and which

[2] `https://www.youtube.com/watch?v=qv6UVOQ0F44`
[3] `https://www.mariowiki.com/Super_Mario_World`

have the characteristics mentioned before, these techniques can save a lot of time for human testers.

When designing a fitness function it is important to take into account that the genetic algorithm may not be able to perform complex behavior in the first generation. Thus every fitness function developed should reward individuals for performing simple behavior that could be performed at random and take it closer to the target behavior. In simpler terms, the fitness function should reward an individual for every step closer to the target behavior, no matter how small it is.

Another thing to take into account is that the input sequences produced by the algorithm are essentially time sequences, and the crossover and mutation functions must be developed taking that into account. Because of this, the sooner an individual gets close to the solution, the better; as it will have more time to do the rest of the steps. This is not always necessary, as shown in the results of the last experiments, but it is something to take into account when setting the time to run the genetic algorithm and the fitness function.

As mentioned, the more precise information the fitness function has about the target behavior, the better. If there is more than one way to get a high fitness, it is more probable that the genetic algorithm performs the easiest way. For example, in site 2 of Hawkthorn's platform experiments, where the fitness function was designed so the individuals produced stayed as close as possible to the starting position of the sequence; the individuals instead stayed in a still platform near the moving platform.

## 6.5 Future Work

There is a lot of work that can be done from what was created in this thesis. First, the simplest way to continue the work is to complete the wrapper and catch all of LÖVE events so the wrapper can use them. More types of inputs can be implemented in the wrapper and the input sequence encoding, including mouse, joystick and touch inputs.

The genetic algorithm can also be extended to include these types of inputs in order to extend the amount of games compatible with the software developed. Joystick inputs should be the easiest one to implement as most of them are just more buttons. Mouse and Touch inputs on the other side can be more complex, as these inputs can come from any position of the canvas used by a game.

More experimentation can be done by editing the current encodings and creating new encodings to represent input sequences. Following the same line, new crossover and mutation functions can be implemented for existing individuals in order to get a better understanding of the possible use for each. Finally other GA concepts can be applied to the software developed, like speciation for example.

Some updates to the genetic algorithm software can be done so it would be easier to use for developers. These updates can include a basic graphic interface for editing the software parameters without editing the files directly. It can also include a feature to import a game so the software makes the installation by itself instead of editing the files of the game.

Finally the reduction feature can be validated and extended so input sequences become more accessible to users. The current implementation uses independent individuals in order to perform the reduction operations. With this same objective, input sequence visualization and editing tools could be developed. All of these features would greatly increase the value of the software in its use for video game testing.

# Bibliography


[1] Approximation algorithm. https://en.wikipedia.org/wiki/Approximation_algorithm. Accessed: 2020-09-01.

[2] A. Brindle. Genetic algorithms for function optimization. 1980.

[3] P Christensson. Lua definition. https://techterms.com/definition/lua. Accessed: 2020-03-25.

[4] Tim Crosby. How video game testers work. https://electronics.howstuffworks.com/video-game-tester2.htm. Accessed: 2020-05-28.

[5] Kenneth Alan De Jong. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems.* PhD thesis, USA, 1975. AAI7609381.

[6] Patrick Dickinson. Instant replay: Building a game engine with reproducible behavior. https://www.gamasutra.com/view/feature/131466/instant_replay_building_a_game_.php, 07 2001. Accessed: 2020-05-28.

[7] Python Software Foundation. What is python? executive summary | python.org. https://www.python.org/doc/essays/blurb/. Accessed: 2020-03-25.

[8] Alex S. Fraser. Simulation of genetic systems by automatic digital computers i. introduction. *Australian Journal of Biological Sciences 10(4) 484 - 491*, 1957.

[9] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Longman Publishing Co., Inc., 75 Arlington Street, Suite 300 Boston, MAUnited States, 1 edition, 1989.

[10] Silicon Graphics. Opengl sdk. https://www.opengl.org/sdk/libs/. Accessed: 2020-03-25.

[11] John H. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, MI, 2 edition, 1975. second edition, 1992.

[12] Sam Lantinga. Simple directmedia layer library. https://www.libsdl.org/. Accessed: 2020-08-05.

[13] LoubiTek. Collision - attacking when coming up from crouch. https://github.com/

hawkthorne/hawkthorne-journey/issues/2456. Accessed: 2020-04-15.

[14] Löve (game engine). Löve (game engine) - wikipedia. https://en.wikipedia.org/wiki/L%C3%B6ve_(game_engine). Accessed: 2020-03-25.

[15] Jim Merrill. Automated testing for league of legends. *Riot Games*, 2 2016.

[16] Johannes Pfau, Jan Smeddinck, and Rainer Malaka. Intelligent completion of adventure riddles via unsupervised solving. In *Automated Game Testing with ICARUS*, pages 153–164, 10 2017.

[17] Brian Provinciano. Automated testing and instant replays in retro city rampage. GDC, 2015.

[18] John Sturtz. https://realpython.com/python-dicts/#:~:text=Dictionaries%20are%20Python's%20implementation%20of,key%20to%20its%20associated%20value., August 6, 2018. Last Modified: 2020-05-05. Accessed: 2020-06-03.

[19] Marc Tanti. Fitness function for multi objective optimization / mapping vectors/tuples to numbers. https://geekyisawesome.blogspot.com/2013/06/fitness-function-for-multi-objective.html. Accessed: 2020-04-02.

# Appendix

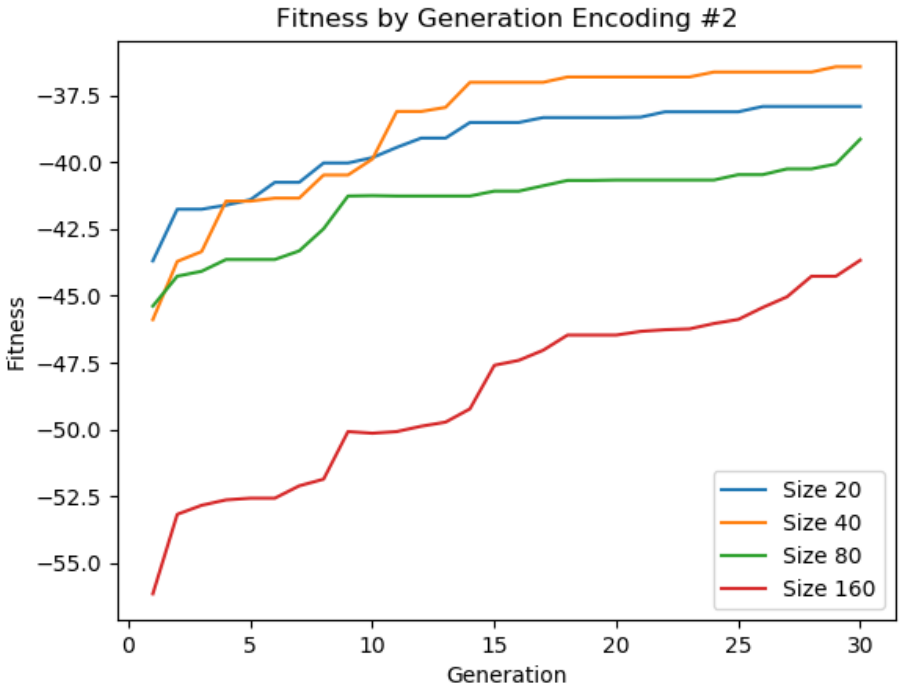


Figure 6.1: Comparison of encoding #2's individuals varying the gene amount along the generations.

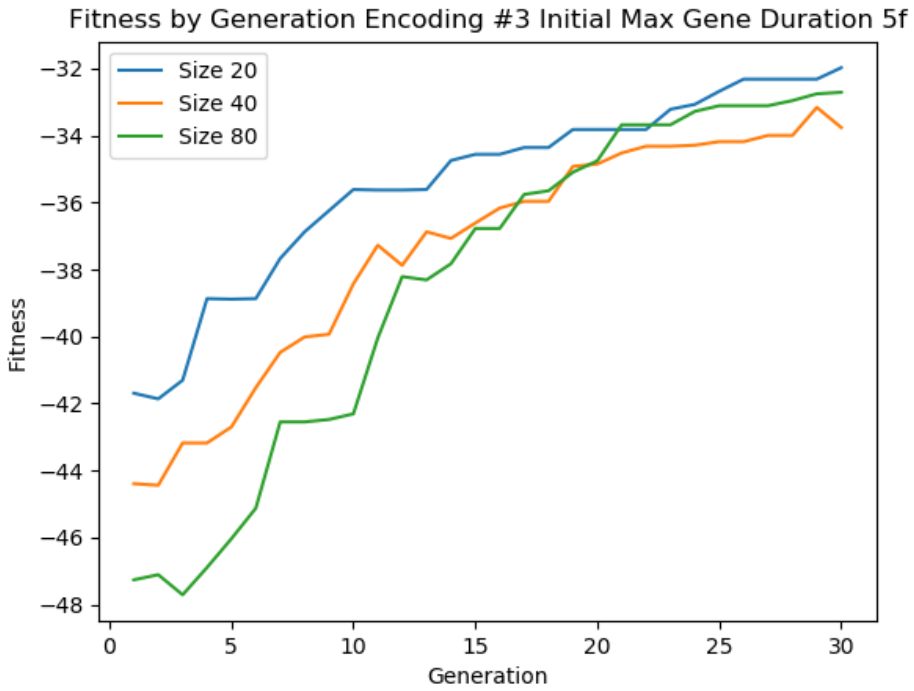


Figure 6.2: Comparison of encoding #3's individuals varying the gene amount with a max duration of 5 frames per gene.

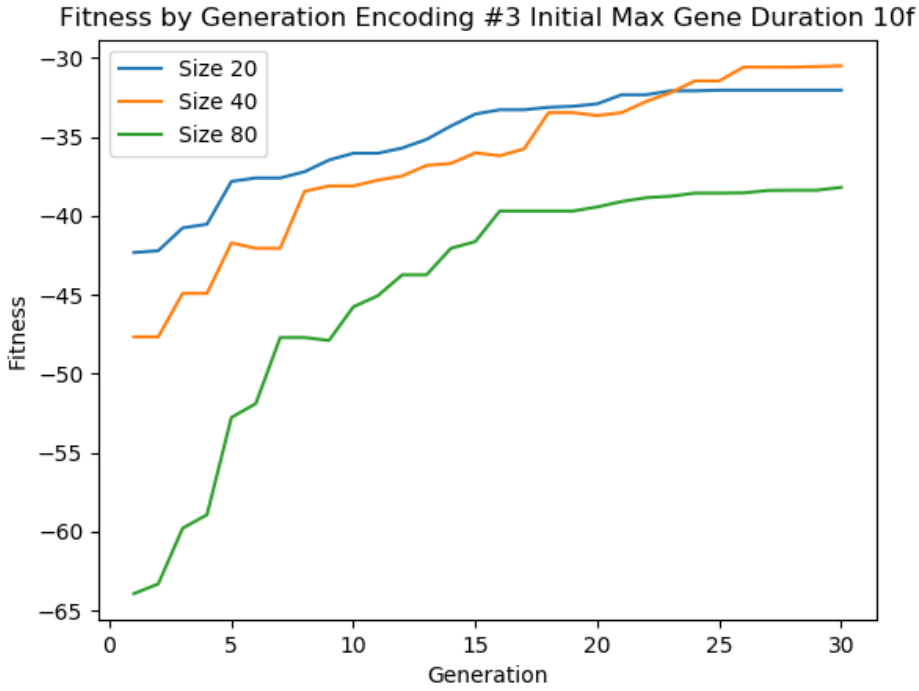


Figure 6.3: Comparison of encoding #3's individuals varying the gene amount with a max duration of 10 frames per gene.

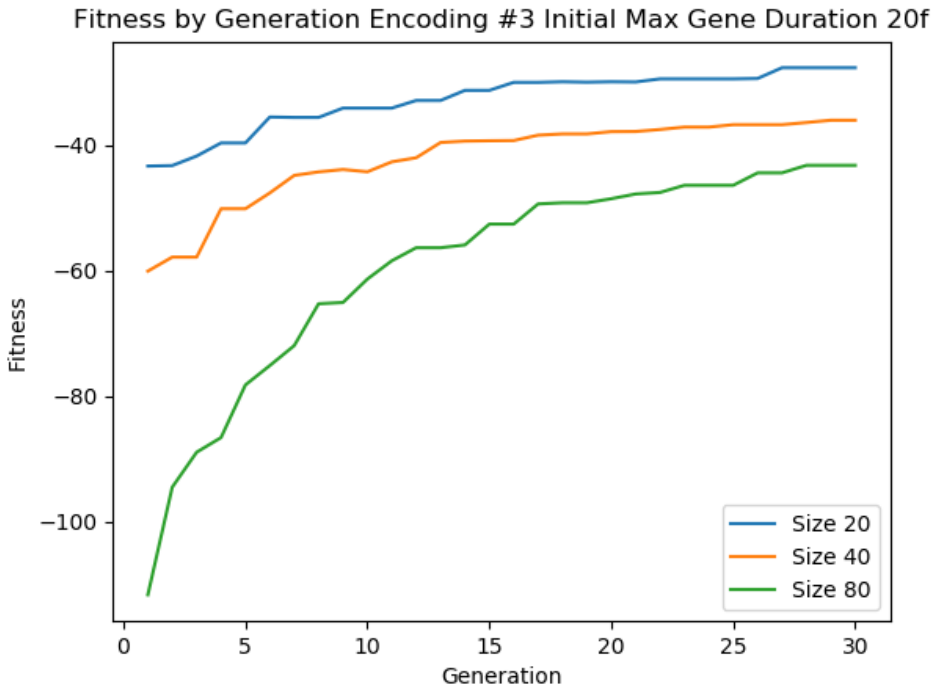


Figure 6.4: Comparison of encoding #3's individuals varying the gene amount with a max duration of 20 frames per gene.

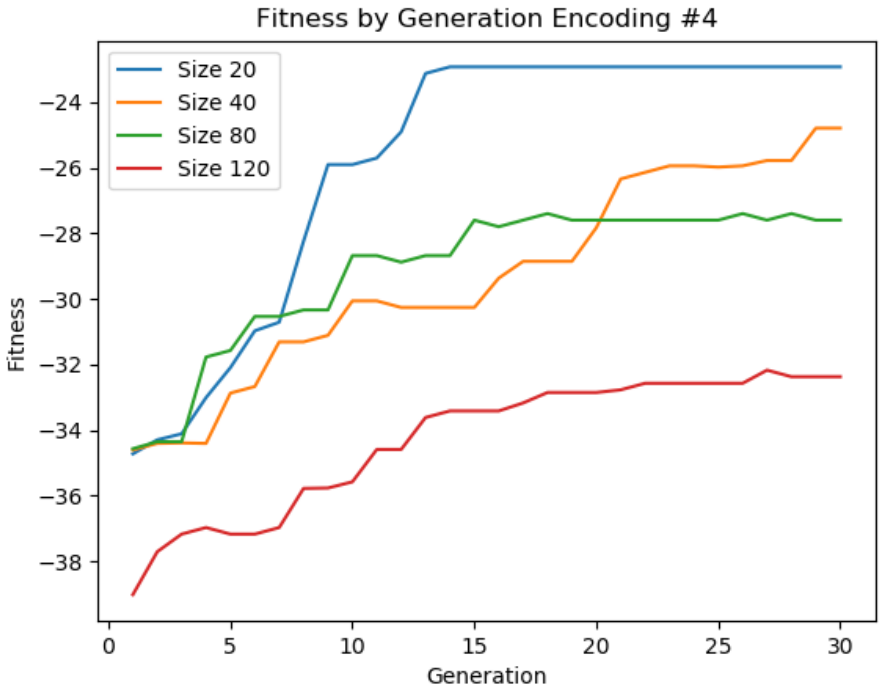


Figure 6.5: Comparison of encoding #4's individuals varying the gene amount.

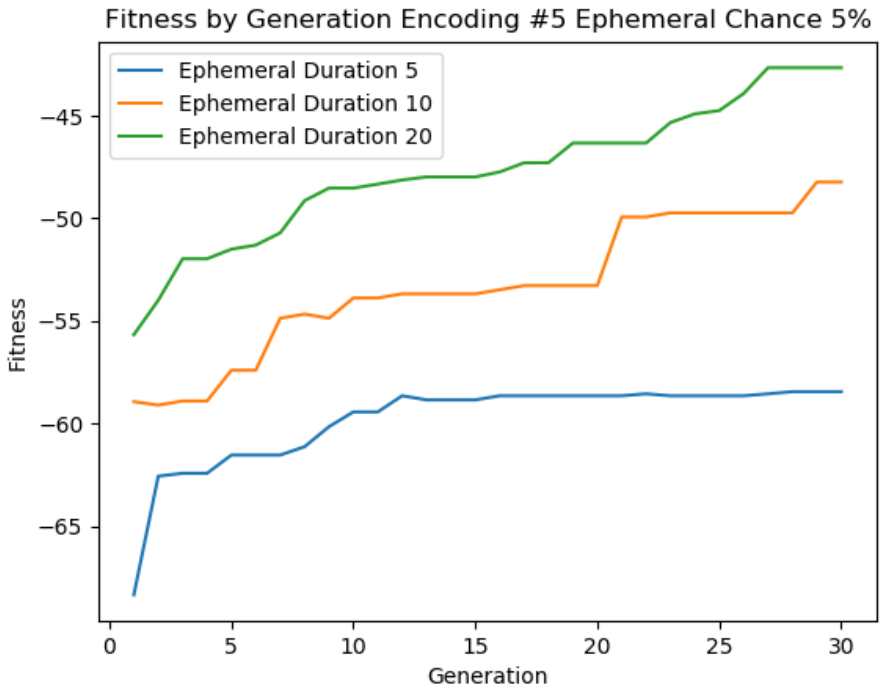


Figure 6.6: Comparison of encoding #5's individuals varying the ephemeral gene duration given a 5% of frames given by ephemeral individuals approx.

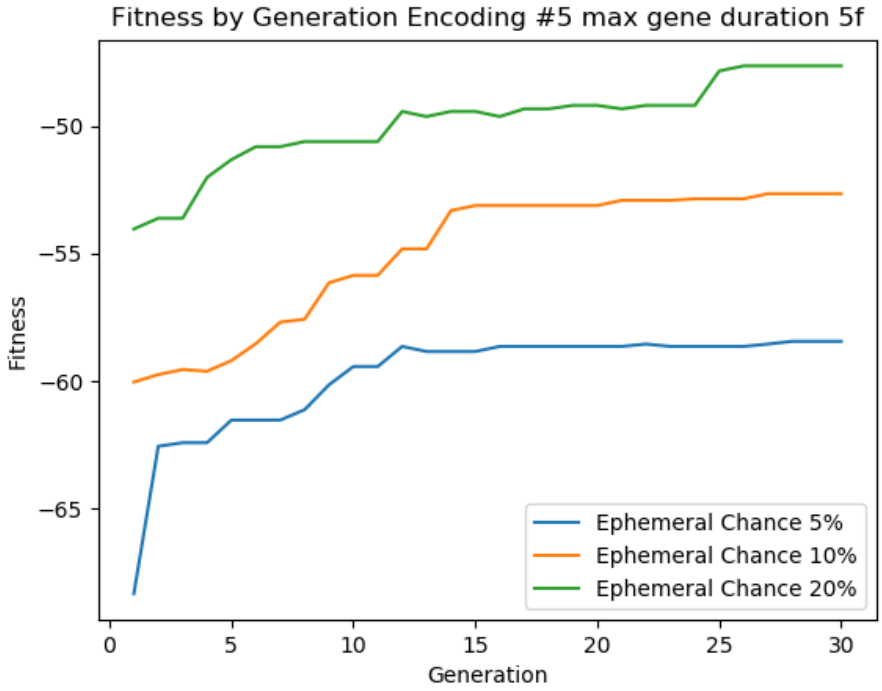


Figure 6.7: Comparison of encoding #5's individuals varying the ephemeral gene chance given a maximum ephemeral gene duration of 5 frames.

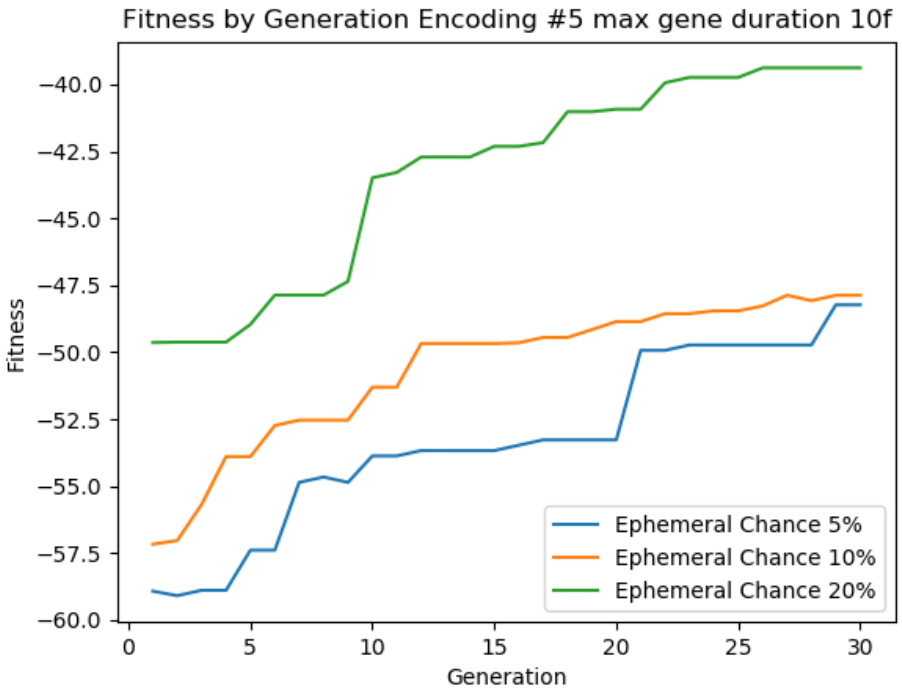


Figure 6.8: Comparison of encoding #5's individuals varying the ephemeral gene chance given a maximum ephemeral gene duration of 10 frames.

$$sigmoid(x) = \frac{1}{1+e^{-x}}$$

Figure 6.9: Definition of the sigmoid function.

```
// define the first generation as generation 0
t := 0;
Compute initial population B[0];

WHILE stopping condition not fulfilled DO BEGIN

   // selection
   FOR i:=1 TO n DO
      select bi,t+1 from B[t]

   // crossover
   FOR i:=1 TO n STEP 2 DO
      with probability pc perform crossover of bi,t+1 and bi+1,t+1

   // mutation
   FOR i:=1 TO n DO
      with probability pm eventually mutate bi,t+1

   t:=t+1;
END

return best bi from B[t]
```

Figure 6.10: Pseudo code of the whole process of the genetic algorithm. $n$ represents the number of individuals in the population per generation. In this example the stopping condition is not specified but it is typically reaching generation $x \in \mathbb{N}$ or getting an individual whose fitness is more than $k \in \mathbb{R}$.

Figure 6.11: Part 1 of a progression of images that shows Chai's Special Dash Attack in Zabuyaki.

Figure 6.12: Part 2 of a progression of images that shows Chai's Special Dash Attack in Zabuyaki.

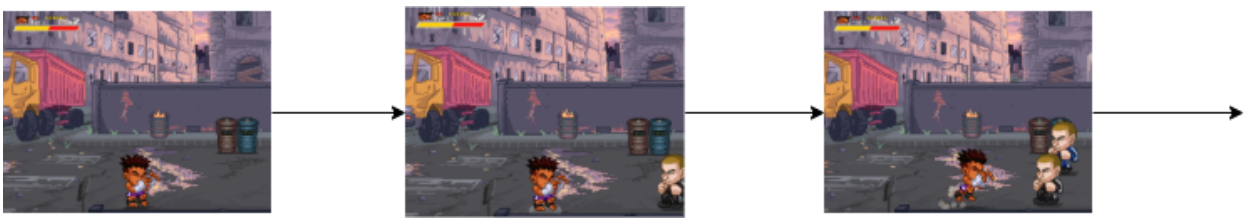

Figure 6.13: Part 1 of a progression of images that shows Chai's Charged Dash Attack in Zabuyaki.

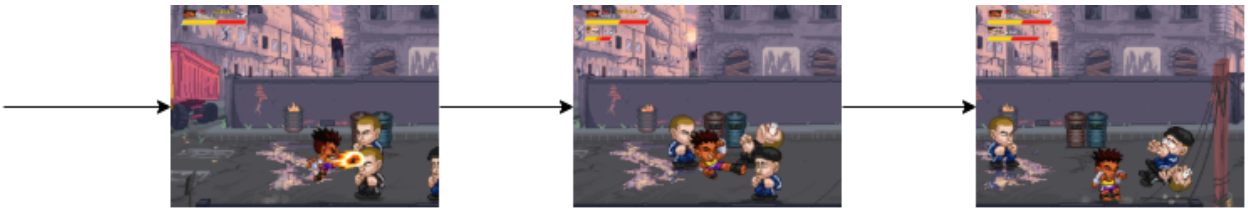

Figure 6.14: Part 2 of a progression of images that shows Chai's Charged Dash Attack in Zabuyaki.