Eötvös Loránd University

Faculty of Informatics

Department of Numerical Analysis

Solving the Climb: An Evolutionary Algorithm Approach to Achieving Upward Movement

Supervisor:

A H M Sajedul Hoque

PhD Candidate

Author:

Alaaraj Judeh

Computer Science BSc

Budapest, 2025

EÖTVÖS LORÁND UNIVERSITY

FACULTY OF INFORMATICS

Thesis Topic Registration Form

Student's Data:

Student's Name: Alaaraj Judeh A.J.

Student's Neptun code: HE84W3

Educational Information:

Training programme: Computer Science BSc

I have an internal supervisor

Internal Supervisor's Name: *A H M SAJEDUL HOQUE*

Supervisor's Home Institution: ***Department of Numerical Analysis***

Address of Supervisor's Home Institution: ***1117, Budapest, Pázmány Péter sétány 1/C.***

Supervisor's Position and Degree: *PhD Candidate*

Thesis Title: Solving the Climb: An Evolutionary Algorithm Approach to Achieving Upward Movement

Topic of the Thesis:

(Upon consulting with your supervisor, give a 150-300-word-long synopsis os your planned thesis.)

This thesis explores the application of evolutionary algorithms to solve the task of upward movement in a game environment, achieving upward progression through iterative, adaptive techniques. Inspired by principles of natural evolution, the evolutionary algorithm simulates a process of selection, mutation, and reproduction to gradually improve the ability of an agent to reach higher levels within a simulated environment.

The thesis presents a detailed implementation of the algorithm, starting from the initial random generation of potential solutions to the gradual refinement across generations based on a fitness function designed to reward upward progress. Key challenges such as defining suitable mutation rates, crossover techniques, and fitness evaluation criteria are addressed to ensure the algorithm's effectiveness in completing the task.

The game features a character who can jump upward and move side to side, aiming to reach the top of a stage from the very bottom. The challenge lies in navigating obstacles and platform placements, requiring strategic movement to complete the climb.

Budapest, 2024. 10. 15.

Tale of Contents

Chapter 1

Introduction

1.1 Introduction

Charles Darwin popularized the idea that given enough time, a singular species could evolve and change based on the factors around it [2]. His most popular example, the Galápagos island finches, showcased this idea perfectly. He noticed how finches in the separate islands had differently shaped beaks, suited for consuming the food found in that island. Either having long slender beaks for searching through flowers and catching insects, or harder and broader beaks which helped the birds crack tough nuts.

1.2 Motivation

Though these birds came from the same ancestor, it appeared that through time they have changed! I aim to simulate this process of change and adaptation through a virtual environment with two dimensional bots.

1.3 Problem Statement

To simulate the process, it is necessary to create an environment where agents are able to play, learn, and adapt as a population through the use of an evolutionary algorithm. The core problem is in the designing of such an environment, the details of the evolutionary algorithm, and the creation of agents with some cognitive ability.

1.4 Objectives

Similar to how the birds with the best beaks are able to get the most food, which allows them to have a higher chance to spread their genetics and traits, the best performing bots get to spread their genetics and their traits are more likely to appear throughout the next generations. This whole process occurs in a simple platformer, where the bots are able move left or right, choose a jump direction, and jump strength. The goal is for the bots to be able to jump correctly and go up from platform to platform.

Chapter 2

User Documentation

This chapter will act as a guide and introduction to the specifics of my program. It is structured in a way so that an average user will find the guide useful, not just avid programmers.

2.1 Description

My program in essence is a two-dimensional platformer. A platformer is a type of video game where the player's objective is to go from point to point by jumping and running. However, in my program, I aim to create and evolve a population of artificial intelligence agents which are capable of succeeding at the game. The objective of my game is quite simple, the higher the player can go, the better!

I allow a large population of agents to play the game for a certain period of time then choose among the best performers to repopulate the next generation. Similar to how species evolve in our real world, my program chooses the best performing agents through different selection methods. This same cycle is what will be constantly repeated, and hopefully over some time, results in agents who play the game better.

2.2 Target Audience

My work is designed and aimed towards beginners, students, or any other curious people who would like to explore the idea of evolutionary algorithms. It serves as a visual and interactive way to understand how a population could evolve and

change over time. Allowing the user to witness in real time the mistakes a population could make, how different and varied their choices may be, etc.

You don't require any advanced knowledge of machine learning or game development to run the program, for it works with a simple example straight away. However, a more advanced user, more familiar with the idea of training a population and programming, would also have a lot of fun with the program by altering the source code itself.

2.3 Main Methods and Tools

- Python [6]: The virtual environment, the ai agents, and the evolutionary algorithms are all built using Python. Python is a high-level object-oriented programming language known for its simple syntax and user-friendly readable code.
- Pygame [7]: Pygame is a Python library which is used to create mainly two-dimensional games. It is quite useful, helping me create the screen, user inputs, game objects to avoid lengthy collision math, and more.
- PyTorch [8]: The ai agents need a way to think. PyTorch is a powerful machine learning library which I used to create the agents' brains. They are forward feeding neural networks; which will be explained later with more detail, but in essence, they are functions which take a set of inputs and give me a set of outputs determining the agents' moves.
- Matplotlib [5]: A plotting library, helping to visualize all kinds of data in Python using charts, graphs, and more. I personally use it to create a graph showcasing simulation data at the end of a run.
- unittest [10]: Python's built in testing framework, also known as PyUnit, follows the xUnit style of testing. Proper testing ensures correct, consistent behavior and that functions like the selection methods return expected outputs.

All of these tools and libraries are combined together to form my program and make it work in a seamless, user-friendly, and readable manner.

2.4 System Requirements

- It is true that the program is rendering objects, however, a dedicated graphic processing unit (gpu) is not necessary. A central processing unit (cpu) with integrated graphics is more than enough.
- Since the game requires user inputs, a keyboard and mouse should be used along with a monitor to display the program.
- The storage requirements are similar for the three major operating systems, Windows, macOS, and Linux. The program's source code and assets are quite small, requiring less than a megabyte of data; however, it will not run without the additional libraries and Python. Since we need a Python version of at least 3.9, that will take around $200 - 300$ Megabytes of storage once installed on your system. The additional libraries required to run the program will also take up a combined $1.3 - 1.5$ Gigabytes of data. The reason it is a bit large is because of the PyTorch library which is quite a large and powerful one. Therefore, I recommend at least around 2 Gigabytes of free space to be safe before starting the installation process.

2.5 Installation Guide

- After a successful download of the program's source code, you must download the required libraries to run the program.
- The first step of course is to download python if you don't have it already or your version is too old. Since one of the libraries I am using, PyTorch, requires a python version of at least 3.9, that is the minimum you will need to be able to run the game.

- Python 3.9 comes with pip installed automatically, which makes the rest of the installation straightforward.
- After navigating to the root folder of the program I highly recommend creating a virtual environment first. Here are the steps to do that and installing the requirements:
    - *python -m venv venv*
    - Activating the virtual environment is different for Windows and for macOS/Linux
    - The command for Windows is: *venv\Scripts\activate*
    - The command for macOS and Linux is:

      source venv/bin/activate
    - Installing the required libraries with pip is simple. The folder you downloaded for the whole program should include a requirements.txt file, we use this file with the command:

      pip install -r requirements.txt
    - This downloads the libraries listed in the requirements.txt and all the libraries that they also require to run inside the virtual environment which exists inside the same root folder. The use of the virtual environment is useful because it avoids breaking any other programs that might be using different versions of the same libraries.
    - To leave the virtual environment, it is the same command for the different operating systems: *deactivate*
- If you do not want to create a virtual environment but instead are fine with installing the libraries globally, you follow this simple step:
    - Navigate to the root folder of the program and run the command: *pip install -r requirements.txt*

2.6 Running the Program

After a successful installation, open the project in an editor or simply navigate to the installation location through a terminal. If you have installed the libraries in a virtual environment, make sure you activate the environment before you run the program or it will claim that you are missing or it does not recognize some modules. You are able to run the program with the following command, whether you are in the activated virtual environment or you have installed the libraries globally:

python -m source.control.controller

This command ensures the program is ran as a module and links all the components together instead of a simple python script. The following is the first screen shown to the user after the program runs:

Figure 2.1: Main Menu

This screen, shown in figure 2.1, serves as the main menu and presents the user with various options. The user is able to interact with the buttons by moving their mouse over them and pressing left click. Other than "Quit", which is self-explanatory, I will proceed to showcase what each of the buttons do, starting from "Casual Play".

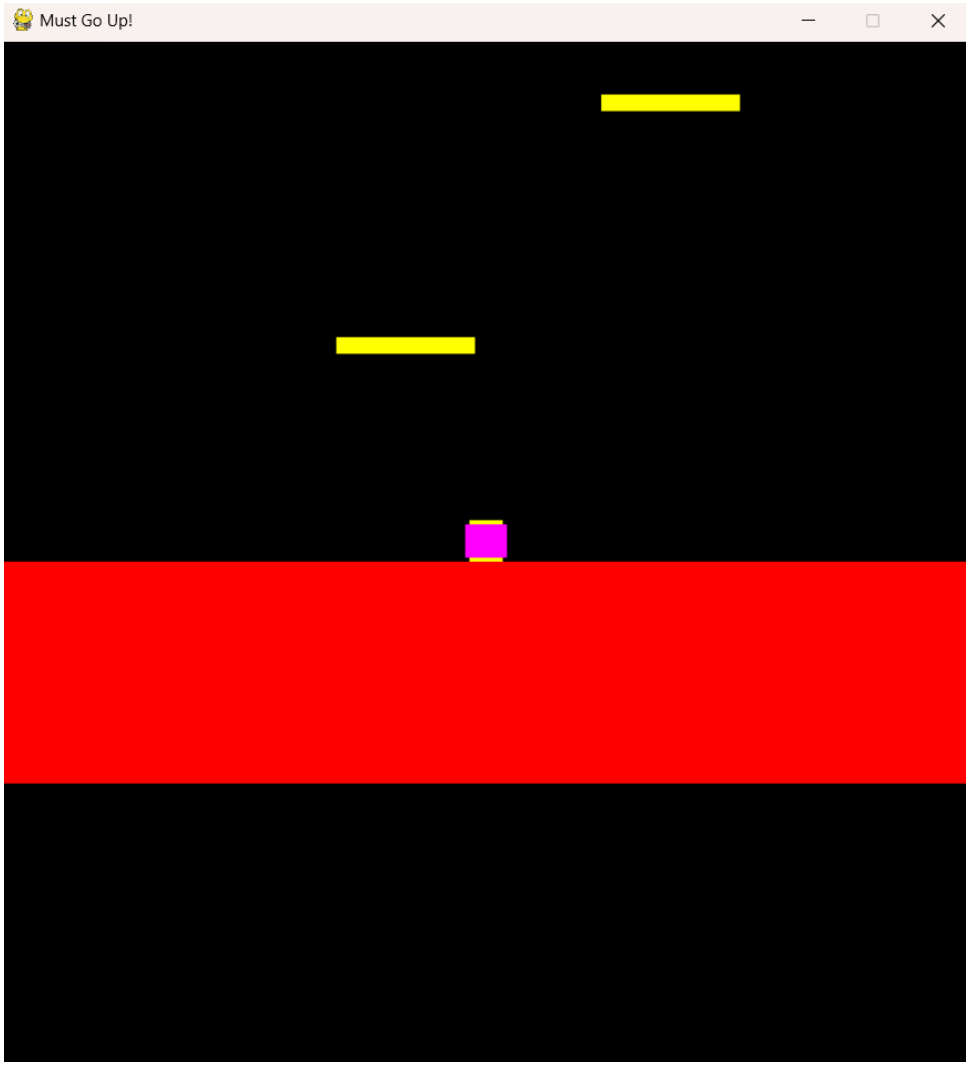


Figure 2.2: Casual Play

Figure 2.2 shows what happens after clicking "Casual Play", the user is presented with this screen, or something very similar since the platforms' positions are randomized each time. This mode exists as a showcase of the game's mechanics to understand the two-dimensional environment that that AI agents will attempt to learn.

By pressing either the left or right keys, the user is able to move in each direction respectively. Holding the space bar initiates charging a jump. While holding the jump button the user is locked into place, releasing the key will allow the user to jump directly up. The height of the jump depends on the length the user held down the space bar, to a limit of course. Holding either the left or right keys along with releasing the

space bar after charging a jump will result in a diagonal jump in that direction. Clicking "Q" on your keyboard will take you back to the main menu.

Clicking the option for "Instructions" will showcase this screen:

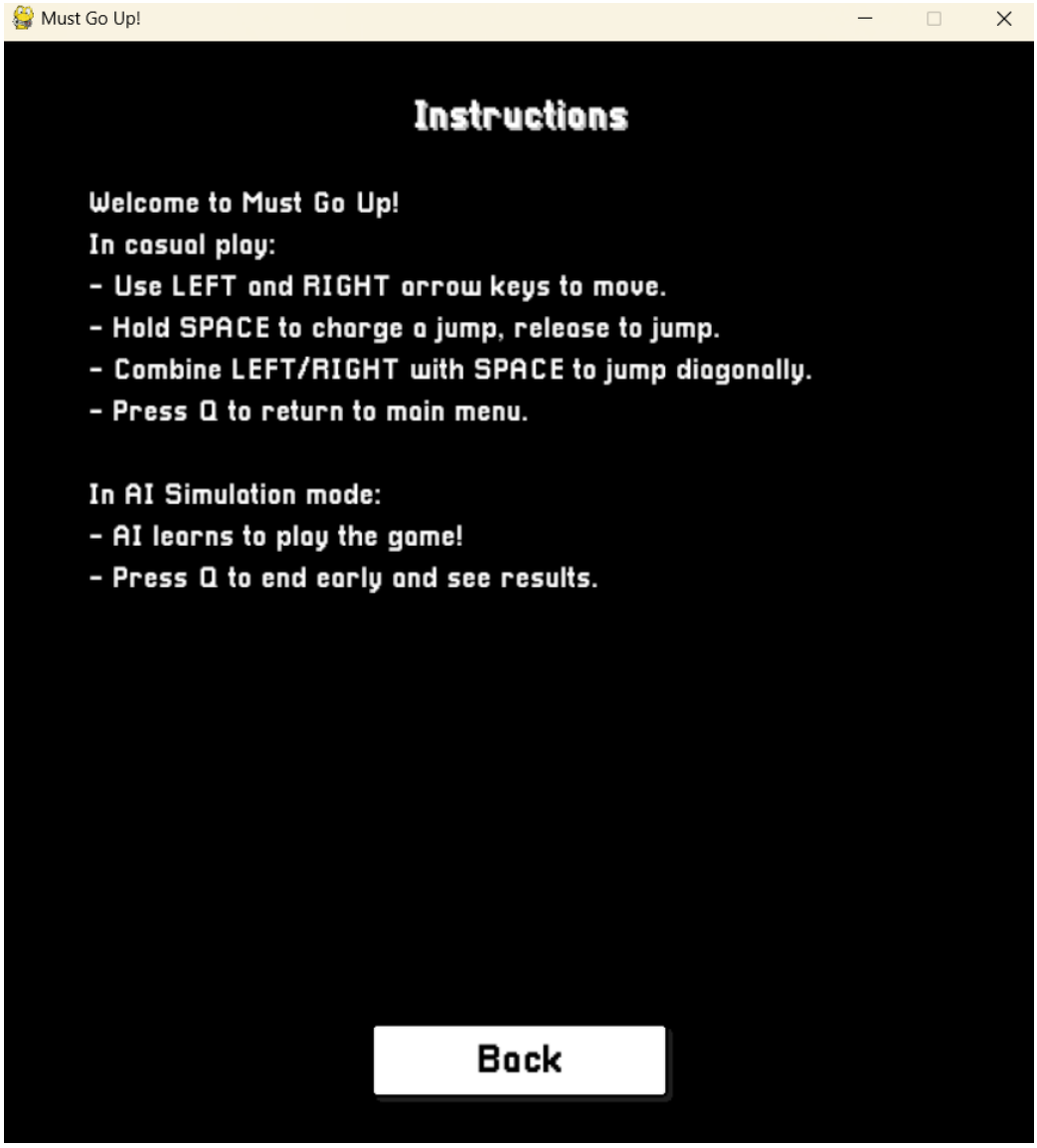


Figure 2.3: Instructions

This screen serves as a guide, you can see it in figure 2.3, for the controls of the game and a simple explanation for what is happening in each mode. Pressing left click on the "Back" button will take you back to the main menu.

Pressing "AI Simulation" from the main menu will take you to the following screen:

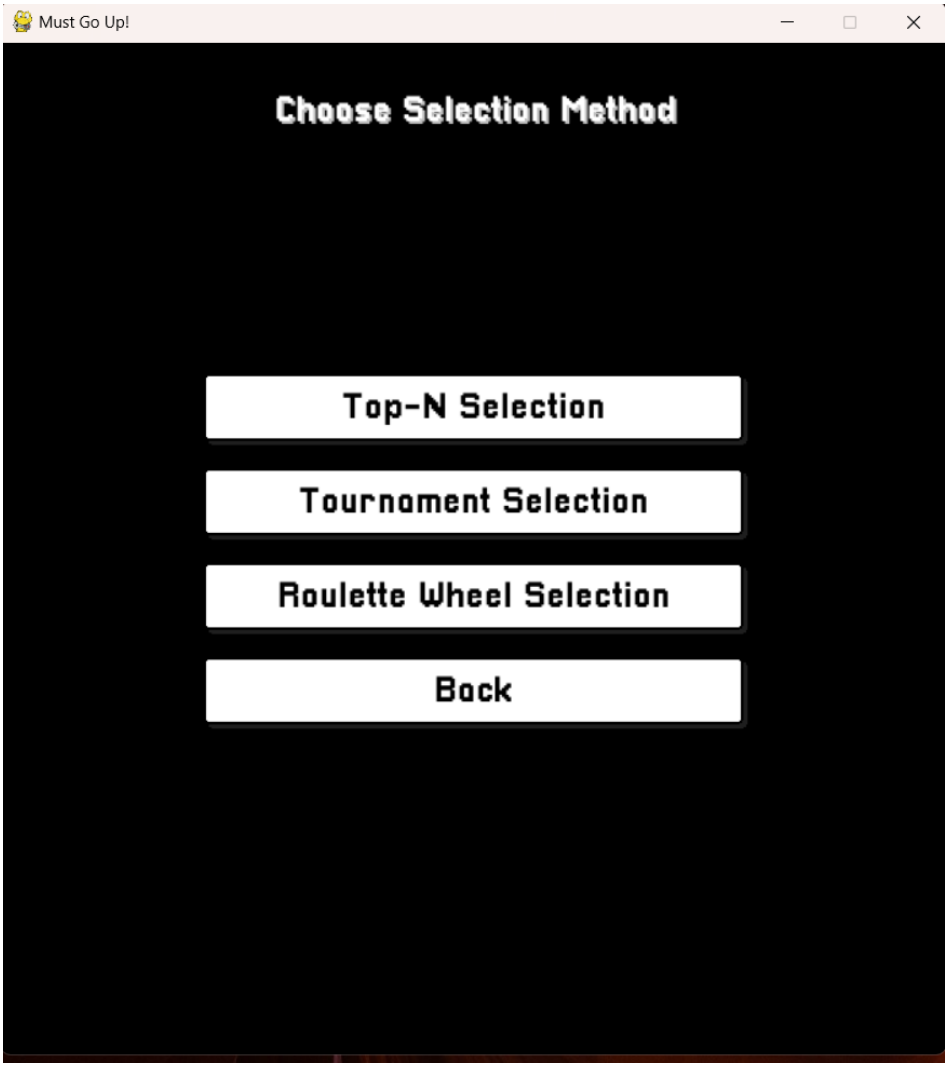


Figure 2.4: Choose Selection Method

Pressing the "Back" button will take you back to the main menu. The other three options shown in figure 2.4, "Top-N Selection", "Tournament Selection", and "Roulette Wheel Selection", are the three selection types the user is able to choose from before starting the simulation. The meaning and differences between the choices will be discussed later. Clicking any of the choices starts the simulation and the following screen is shown:

Figure 2.5: Simulation

The user will see something similar to this screen in figure 2.5 at the start of the simulation. Of-course, it will not be the exact same since the platform positions are randomized to a degree and the initial colors of the bots is random and they have started moving already. The top left of the screen contains information regarding the current simulation. It showcases the number of the generation reached, and the highest fitness achieved this generation. Fitness will be explained later. From this screen you end the simulation by pressing the "Q" key on the keyboard. If you end the simulation before even reaching the second generation, you will get this screen:

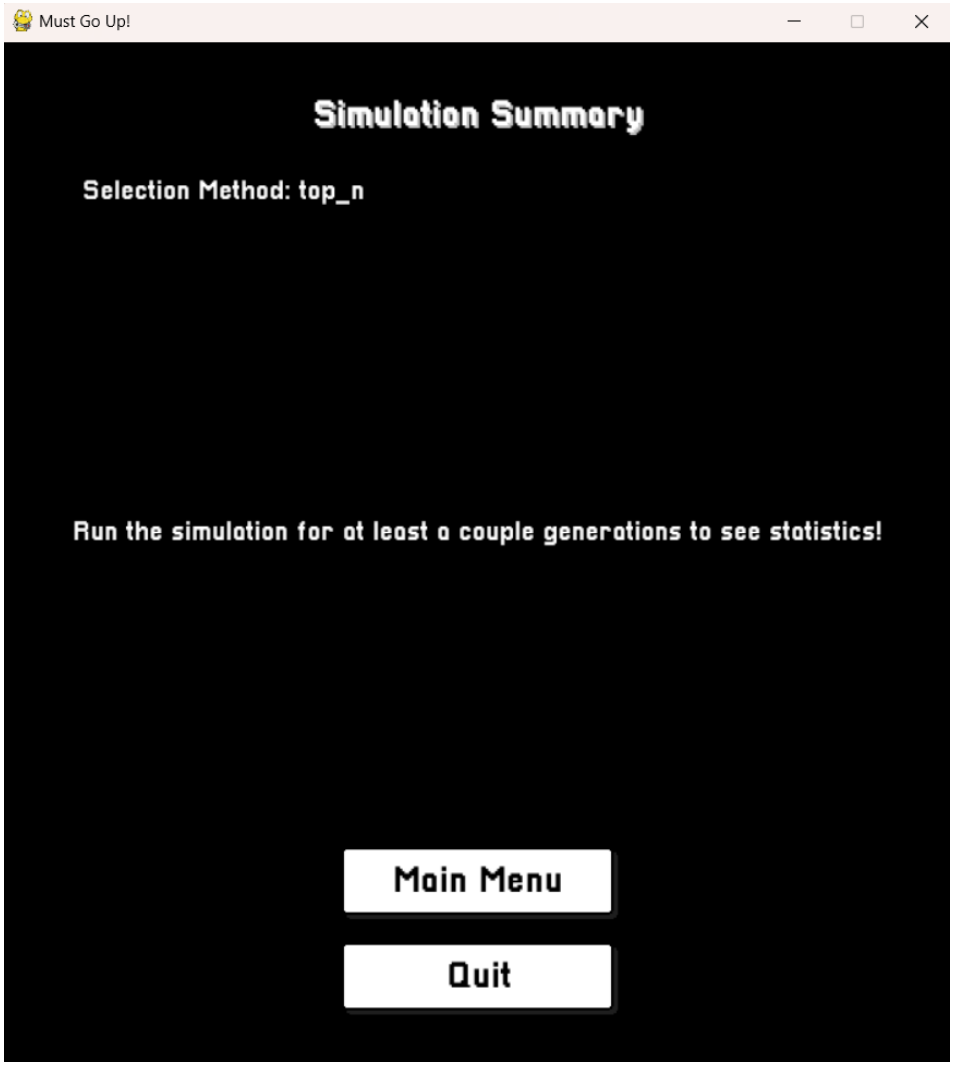


Figure 2.6: Simulation Summary Warning

Figure 2.6 shows how this screen acts as a reminder to let the simulation run for a bit to see some statistics regarding the run. The selection method chosen before the simulation started is written on the top left of the screen. If you let the simulation run for a bit, you will see this screen instead:

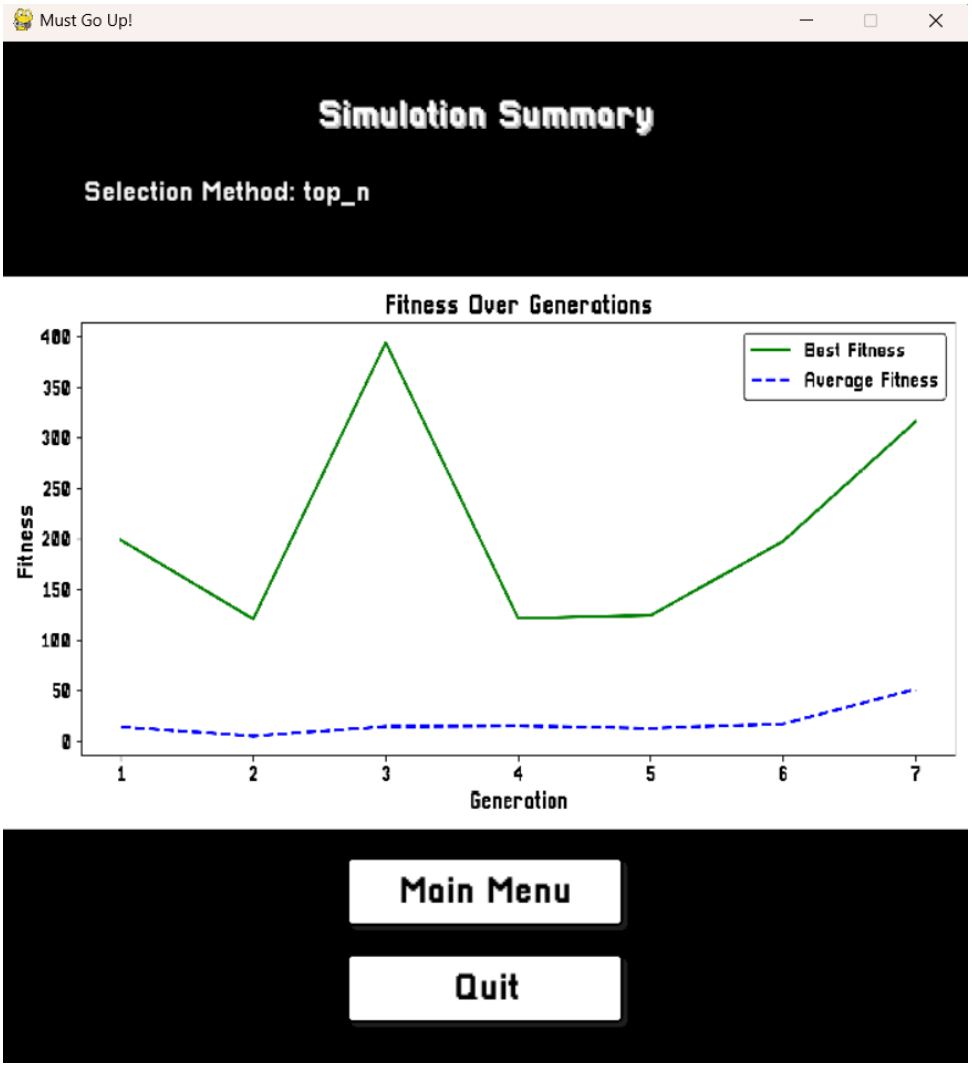


Figure 2.7: Simulation Summary

This is the last possible screen you can reach in the program, shown in figure 2.7. It showcases information regarding the full run of the simulation. The details and possibilities of what this information could be interpreted as is discussed in Testing.

Chapter 3

Developer Documentation

This chapter focuses on the design and details of the program in a programmer's point of view. The idea behind the project, the specifics of the code, and its testing are all discussed.

3.1 Specification:

There are two major parts to my program, the virtual environment the ai agents get trained on, and the ai agents themselves and what manages them. Furthermore, there must exist a menu system to navigate through the program and show results.

3.1.1 Environment:

The game I chose to simulate the process of natural selection among a population using an evolutionary algorithm is a platformer.

- A player must be able to collide with the platforms.
- Gravity applies to the player consistently.
- Player is able to move horizontally left or right.
- Player is able to initiate a jump, straight upward or diagonally.
- A vertical collision from the top of the player to the bottom of a platform stops the upward movement.
- A horizontal collision to the side of a platform causes the player to bounce in the opposite direction.

- Player loses control in the air, he is only able to move or initiate a jump when he is currently standing on a platform. This means that once a jump is initiated, the player's bottom must be colliding with a platform's top before being able to send inputs again.

3.1.2 AI Agents:

The agents conform to all the rules of the game, the only unique or special thing about them is that they are able to choose their own outputs.

- Following all the rules of the game, an agent accepts inputs into its brain, the neural network, and chooses an action. This cycle of choosing an input and deciding an output occurs once every game frame. However, like a normal player, the ai agent must go through with a jump once its initiated and cannot decide a new action till they land again.
- Managing the agents, ending a generation and repopulating the next generation must also be considered.
- There must be a way to judge which agent performed better, in general, this is called fitness. Each agent has this fitness value and this is what helps to decide which gets to spread their genetics to the next generation.
- The "genetics" that are spread to the next generation are the weights and biases of the neural network. Furthermore, like the real world, they must be mutated slightly to simulate natural evolution.

3.1.3 User Interface:

The menus act as a simple UI guide to choose different selection types and initiate a simulation or view instructions. The user is able to interact with the menus through the buttons they contain. They also show the results at the end of a simulation

using a graph. The graph showcases two important statistics, the highest fitness achieved per each generation and the average fitness of all the bots for that generation. The graph is drawn using matplotlib.

3.2 Design

The design of any program is critical because we organize the software's structure so that anyone who reads it can understand what is going on. This helps keep the program maintainable, scalable, and readable. Specifically, in my project, I chose to use the MVC architectural pattern. The architecture is further showcased by the UML use case diagrams.

3.2.1 MVC Architecture

The Model View Control Architecture is a programming design pattern that splits the responsibilities of the program to sections. Though their responsibilities are split, they are still interconnected and work together to make the program whole. This is the design pattern I chose to follow for my program as it helps to break down a large project to manageable parts.

The model contains the program's logic and data. This is where data is manipulated and defined. In the case where the program is a game, this is where player, environment, and other similar things are located.

The view handles everything that the user is able to see. Functions for drawing game sprites, menus, and text are found here.

The control is what manages and connects both the model and view to create the complete program. This is where most of the action happens; the game loop exists here. The controller is what updates the model classes and passes them to the viewer functions to allow it to draw them.

3.2.2 UML Class Diagrams:

The Unified Modeling Language (uml) class diagrams is a way to visually illustrate and represent the different classes and structure of a program and how they are connected with each other. It also showcases some information about the classes' constructors, functions, and attributes they hold. Since my program follows the MVC architectural pattern, I will first showcase a high-level diagram concerning it, then I will show the diagram for each of the components.

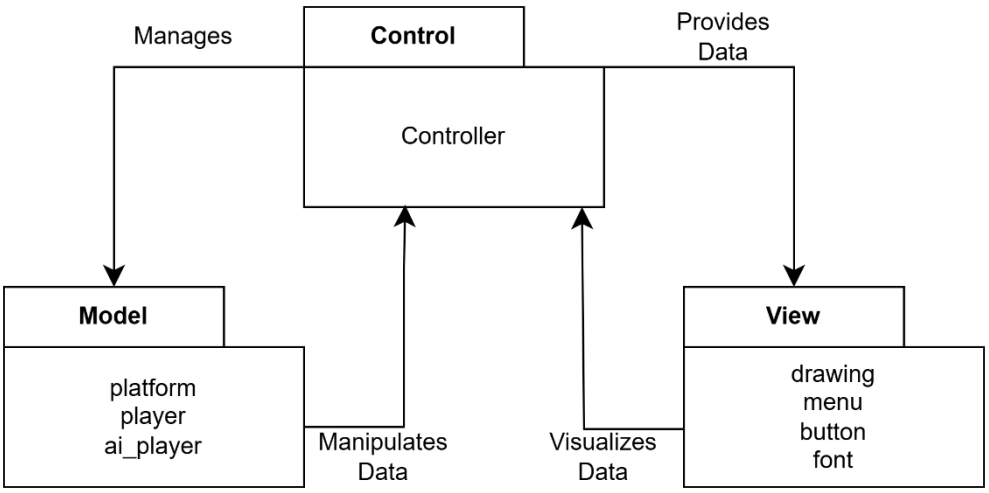


Figure 3.1: MVC Diagram

Figure 3.1 showcases the relationship between the three main components. The model holds the data and objects of our program, the view holds the aspects relating to what the user actually sees, and the controller is the bridge and the connection. The control accepts data from the model, runs the simulation, and is constantly sending data to the view where the view is able to visualize it for us. Since we know how the major components work, I will draw the rest of the components separately, but the connection still holds true.

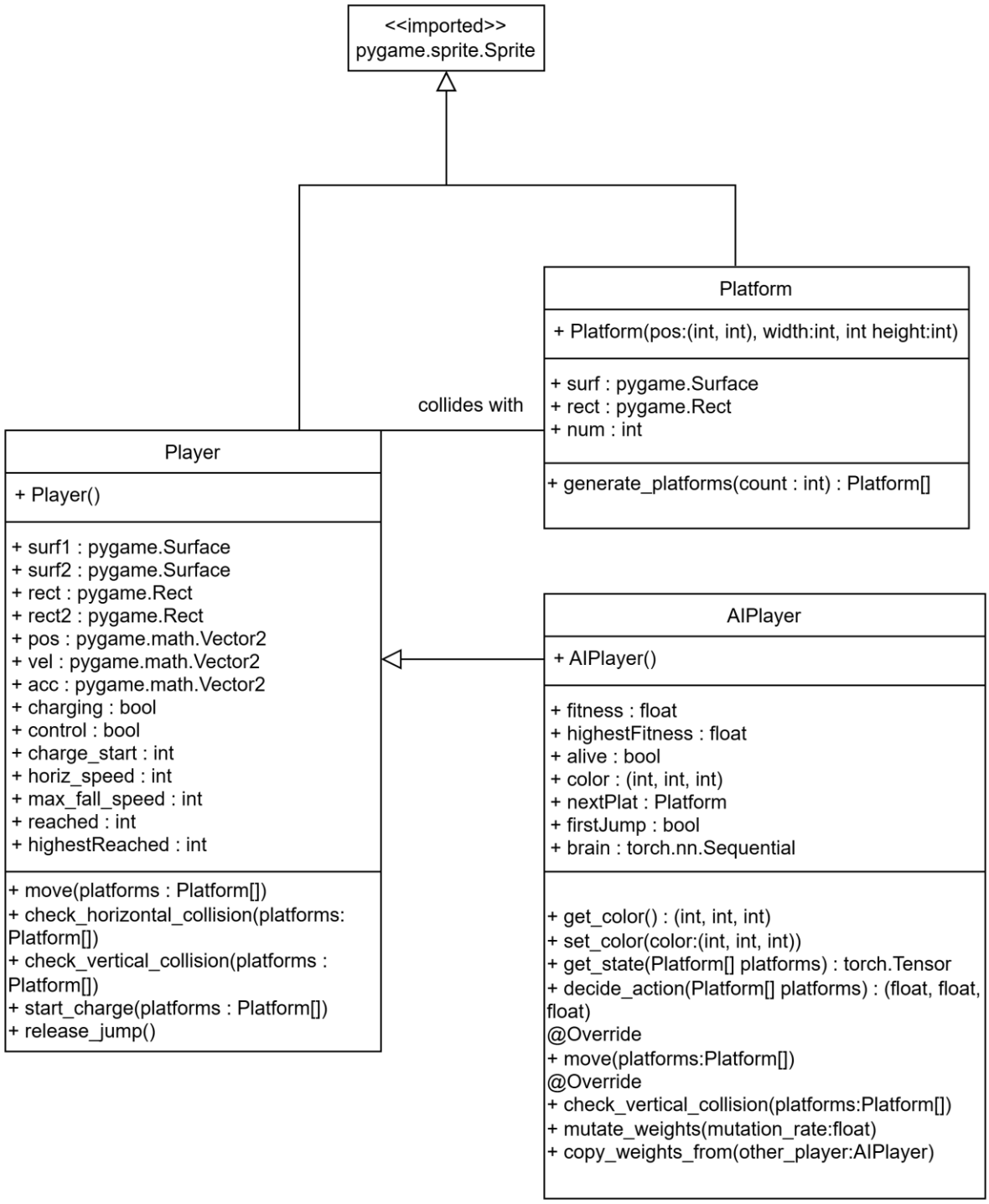


Figure 3.2: Models Diagram

As you can see in figure 3.2, the model holds the classes related to the logic of the game. These are the objects that will be created and needed during the simulation.

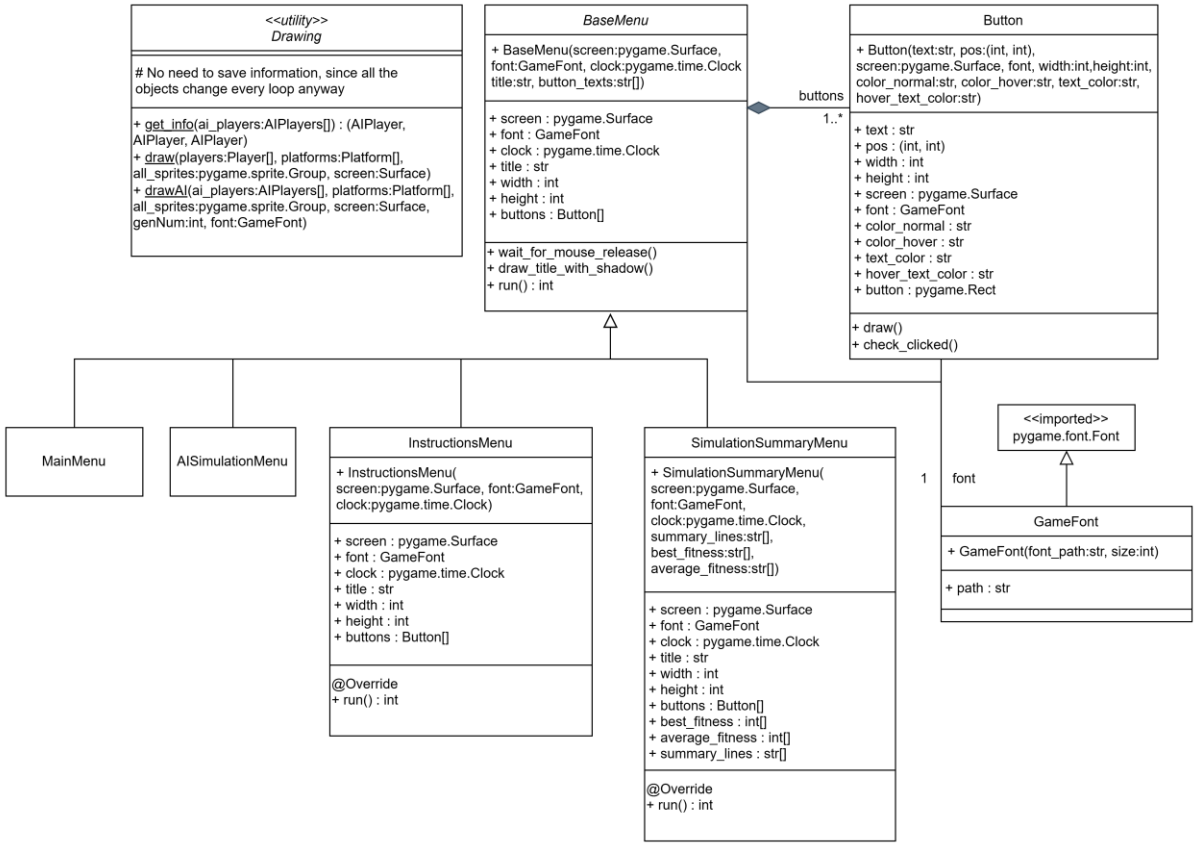


Figure 3.3: Views Diagram

The view, in figure 3.3, holds the classes which are responsible for what the user sees, which means that this is also my user interface. Any new screen the game wants to show the user should be inherited from the BaseMenu screen, since it contains all the logic and buttons for the users to interact with already.

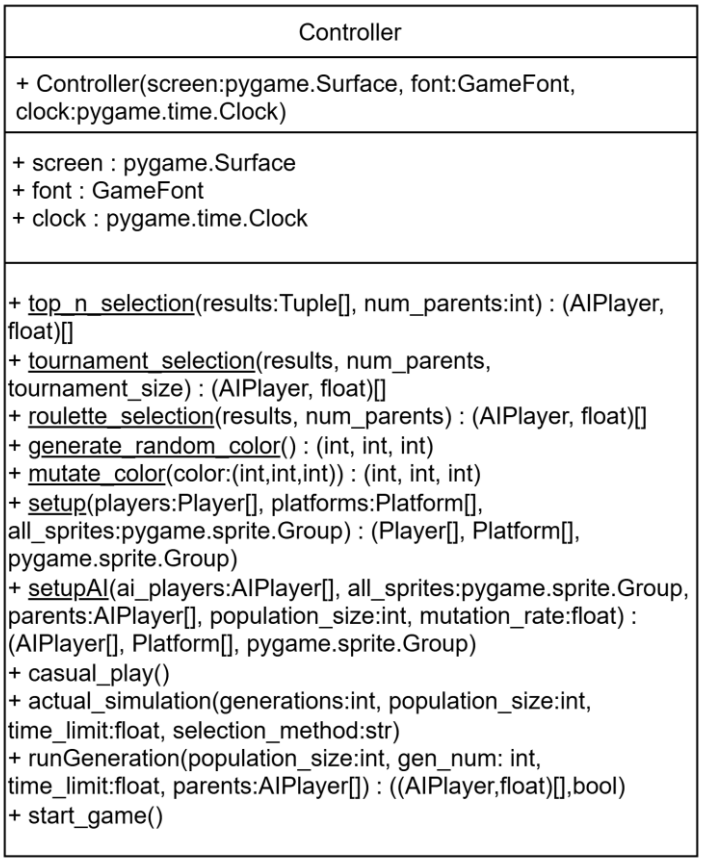


Figure 3.4: Controller Diagram

Figure 3.4 shows the controller and the various methods it contains. This class should be the one responsible for the game loop, creating the objects, passing them to be drawn, etc. To use a real-life example, it is basically the conductor of the orchestra.

3.3 Implementation

Since the code is already split up into separate parts (MVC), I will go through each one and explain how its inner-workings looks like. There are only a couple things outside this MVC architecture that I use in my program. The first is an assets folder which simply includes a sound that plays whenever the user clicks on a button, and a font file. The other is a constants file which is accessible by everyone and just includes constants that do not ever change. The width and height of the screen, number of frames per second, and a Pygame vector are all found there.

3.3.1 Models:

1. Generation of Platforms, their properties and responsibilities:

The platform.py module defines the Platform class which are the platforms that exist for the player or agents to jump on during the game or simulation respectively. The platforms and their random placement act as the obstacle the agents must overcome.

- The Platform class inherits from *Pygame.sprite.Sprite*, this is great because it gives a rectangle to represent the platform with.
- By inheriting from the Pygame class, we also become able to use it along with a Pygame function called colliderect(), which the player or agent will call by passing the platforms as parameters to check collision between them.
- Each platform has an (x, y) position, width and height, and the information about its rectangle representing it.

The platforms are procedurally generated at runtime to create random placements for the player to overcome and also per each new generation to attempt the climb. The spacing and positions of the platforms are random, but still within some bounds to keep the game always possible but challenging.

In the AI Simulation mode, you will notice that a platform will turn white. This is supposed to represent the platform the current highest position agent is trying to jump to.

2. Player's properties, their movements, and construction:

The player.py module defines the Player class, and the AIPlayer class inherits from it.

- The Player class inherits from the same *Pygame.sprite.Sprite* as the Platform class did.
- It holds functions related to the movement of the player and collision checking, and remembers the highest platform it reached.
- I would like to highlight an important aspect of the hitbox of the Player. Here is a photo of a singular player:

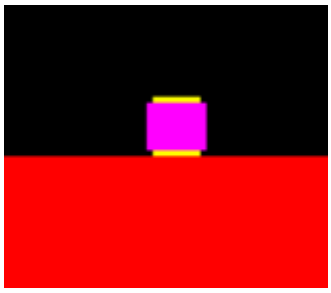


Figure 3.5: Singular Player

You can see that a singular player, shown in figure 3.5, is made up of two distinct rectangles. This is not only a visual design, but instead this is a different approach to detect collisions between objects than normal. If you were to use a singular square to represent the player, you would have to calculate and then check whether you should classify the current collision as a horizontal or vertical collision. To avoid this situation and awkward edge cases, a collision with the yellow rectangle of the player is considered a vertical collision, and a collision with the purple rectangle is considered a horizontal collision.

3. AI Agent's brain, and how all the program ties together:

The ai_player.py module defines the AIPlayer class. It is physically very similar to the player class, and the population we are attempting to teach to play the game is made up of these AIPlayers. There are two important parts I would like to discuss about the AIPlayers that makes them capable of learning how to play the game. The neural network that makes up their brains and the fitness evaluation.

- Neural Network:

Each of the AIPlayers has a brain that consists of a forward-feeding neural network. Neural Networks are quite a large topic and I highly recommend going more into depth about them; however, I will attempt to explain how they work in a simple manner for now and how I implement them in my simulation.

In reality, neural networks are functions. They take a set of inputs and give a set of outputs. In my attempt at training the population, I gave the ai agents' neural network 8 inputs. The agent's x position, y position, the x position of the next platform they must try to jump to, as well as its y position, the agent's velocity in the x direction, as well as the y direction, and the left edges' x position of the next platform the bot must try to jump to, as well as its right edges' x position.

Before I continue my explanation, here, in figure 3.6, is a visual of what the neural network structure of each brain looks like to understand better:

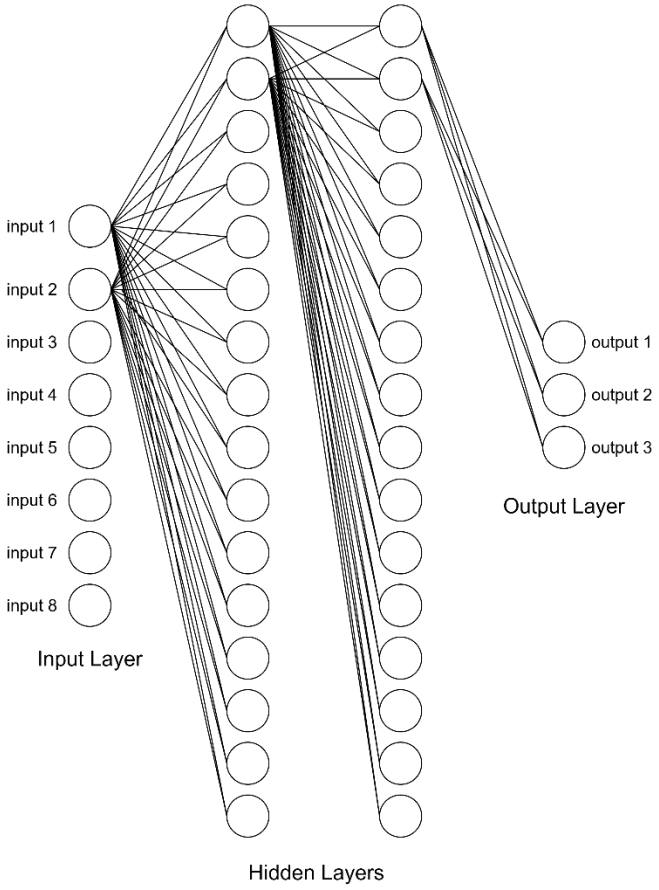


Figure 3.6: Neural Network Diagram

Before moving on I'd like to emphasize that I showcased only the connections of the first 2 nodes (or neurons) in each layer to avoid visual clutter in figure 3.6, but in reality, the same pattern extends all the way down the network in each column of nodes.

The middle nodes are called hidden layers because we as developers do not interact with them directly, they do not know the true inputs and outputs of the program.

The reason there are multiple layers in the network is that it gives it the opportunity to recognize more complex patterns from the inputs. This allows the network a chance to understand and think in steps of increasing complexity. Using my game as an example, the first hidden layer could think about how close a platform is, then in the next hidden layer it could take that information and still think with it. It can still think about whether to move towards the platform or simply start initiating a jump. Then finally, based off of that information, the last output layer gives us outputs indicating the bot's intentions.

From the input layer to the first hidden layer, each of the sixteen neurons in the hidden layer have eight unique weights associated with the connections to the input neurons and a bias. The number calculated by the node is the sum of the weights multiplied by the respective input, plus the bias. Then the activation function is applied to that sum, which decides how fired up or activated the neuron is. There are many activation functions, but for most cases, the Rectified Linear Unit (ReLU) activation function works quite well. It simply takes the maximum of the sum we just calculated and 0, avoiding negative numbers [4]. Simply speaking, it means that if the output of a neuron is negative, it won't activate and does not pass the information forward.

We can represent all the weights related to the first connection from the input layer to the hidden layer with a 16 by 8 matrix. If we put the inputs in a vector and biases as well, we can create a general formula for the calculation of the outputs where "W" is the weight matrix, "b" is the biases vector, and "a" is the input vector. (y = Wa + b). Then the activation function ReLU takes y as an input and returns a as the output ready for the next layer [4]. The next layers calculate the outputs in exactly the same way, the only difference is that there are a different amount of weights, inputs, and biases. This same formula is repeated until we produce the three outputs we need, which will represent movement in the x direction, jump direction, and jump strength.

However, before I can use them I must normalize them to values that make sense in a game setting to realize the bot's intention. I use *torch.tanh()* on the x direction and jump direction to normalize them within values from -1 to 1. Jump strength needs to be a value from 0 to 1 so I use *torch.sigmoid()* to normalize them within that range.

- Mutation Rate

The mutation rate is how much we tweak the weights of the neural network when we mutate them. A lower mutation rate will cause little change to the network, meaning that when we copy and mutate the brain from the parent to the child, the child will not be so different from the parent. If the mutation rate is too high, then the child will not be alike to the parent whatsoever. One of the goals is to find a suitable mutation rate that allows a child to be similar to the parent, but also exhibit new behavior. I personally found out through trial and error that a mutation rate of at least 0.05, but smaller than 0.5 works well in my environment. A mutation rate of a lot more under or above would result in slow or little to no progress. I personally chose 0.05 as the mutation rate to do my simulation testing in, found under the Results and Discussion section.

To discuss why the weights even matter, fitness evaluation must first be explained and understood.

- Fitness Evaluation:

The aim of the bots is to go up by jumping from platform to platform, but how can you communicate that to an AI? The answer is to represent its success with a value, conventionally in similar evolutionary algorithm simulations this value is called fitness. Each of the bots has its own fitness value that is calculated constantly while playing the game, in my version I chose these situations to affect the bot's fitness:

- If it is the bot's first time jumping since it was created in the current generation, I give it fitness. This incentivizes jumping.
- For every new platform the bot jumps up to, it gains fitness.
- Based on the jump strength the bot chooses, I always apply a penalty that goes up the higher the bot jumps. This incentivizes the bot to only jump as high as it needs to and not more.

- How the bot learns:

This is where the whole idea of how the bot learns comes together. Based on this fitness value, we can choose certain bots over the rest of the population to create the next generation with. The way we choose will be discussed later under the section for selection methods. We get their "brains", the neural network, and steal the unique weights they had. We mutate them slightly and create the new generation's brains based off these new slightly mutated weights. Biases are also taken from the parents and mutated, but they are less important in this case than the weights.

The weights determine how the inputs the bots receive from the game transform to the outputs, which are used to decide the bots' moves. The biases help decide how "fired up" or activated a neuron is; therefore, mutating it could help the bots try out new strategies or approaches. This process of choosing the bots based off their fitness values constantly over generations and mutating the brain slightly is what gives the bots the ability to become better at the game. They become suited to returning outputs that result in higher fitness values, which means climbing up higher in the game!

Furthermore, I would like to note something extremely important. The inputs I am giving the neural network, its structure, and the way I am calculating fitness for each bot is only one possible solution out of many to the problem of training the agents. To a non-experienced programmer or someone new to this topic, the program works as a showcase of my specific example, but I highly recommend a more experienced

individual to try their own ideas for fitness evaluations and different inputs to the agents' brains.

3.3.2 Views:

The views are responsible for the visual elements of the game like the user interface and the drawing of the game's sprites. The controller creates and allows the menus to be displayed and ordered correctly from the view. It also gets the sprites and game logic from the Model and passes the results to the view to be drawn once every frame (the game runs at 60 frames a second).

1. Rendering Text

The GameFont class is quite a simple one. I needed the functionality from pygame's own font class, which renders text, but I also wanted to save the font path in the same object.

2. Clickable Buttons

The button module holds the button class which represents the clickable buttons the user can press on the menus.

3. User Interface

The menu module holds an abstract BaseMenu class, which we create various other menus in the game from, like the Main Menu, Instructions Menu, and so on. The menus contain the buttons, draw them, and wait for the user to click on one of them.

4. Rendering the players, platforms, and more.

The drawing module contains the Drawing utility class. This class contains the functions that are able to receive information from the controller class. It takes the

players, ai agents, platforms, and more and draws them on the main screen for the user to see.

3.3.3 Control:

1. Controller and the Evolutionary Algorithm

The controller class handles the connection and information passing between the models and views. It also contains the game loop where the simulation is run, which simulates the Evolutionary Algorithm typical steps of selection, mutation, and replacement [3]. I would like to highlight the selection methods here that belong to the controller class.

- Selection Methods:

A selection method refers to how the bots will be chosen to repopulate the next generation [3]. There are many different kinds of selections, but I chose three main ones that showcase the concepts behind why you would even need many forms of selections.

- Top-N Selection:

```
@staticmethod
def top_n_selection(results, num_parents=5):
    results.sort(key=lambda x: x[1], reverse=True)
    return [(p, fit) for p, fit in results[:num_parents]]
```

Code 3.1: Top-N Selection

Code 3.1 showcases the implementation of the top-N selection. In my experience, this seems to be the selection method with the best outcome according to the fitness evaluation and structure of neural network I chose. This function takes a list of tuples containing a pair of AIPlayers and their fitness. Then it sorts the list based off the highest fitness values and returns the top number of parents you chose. In this selection, we are always choosing the best performing bots.

- Tournament Selection

```
@staticmethod
def tournament_selection(results, num_parents=5, tournament_size=4):
    selected = []
    for _ in range(num_parents):
        tournament = random.sample(results, tournament_size)
        winner = max(tournament, key=lambda x: x[1])
        selected.append((winner[0], winner[1]))
    return selected
```

Code 3.2: Tournament Selection

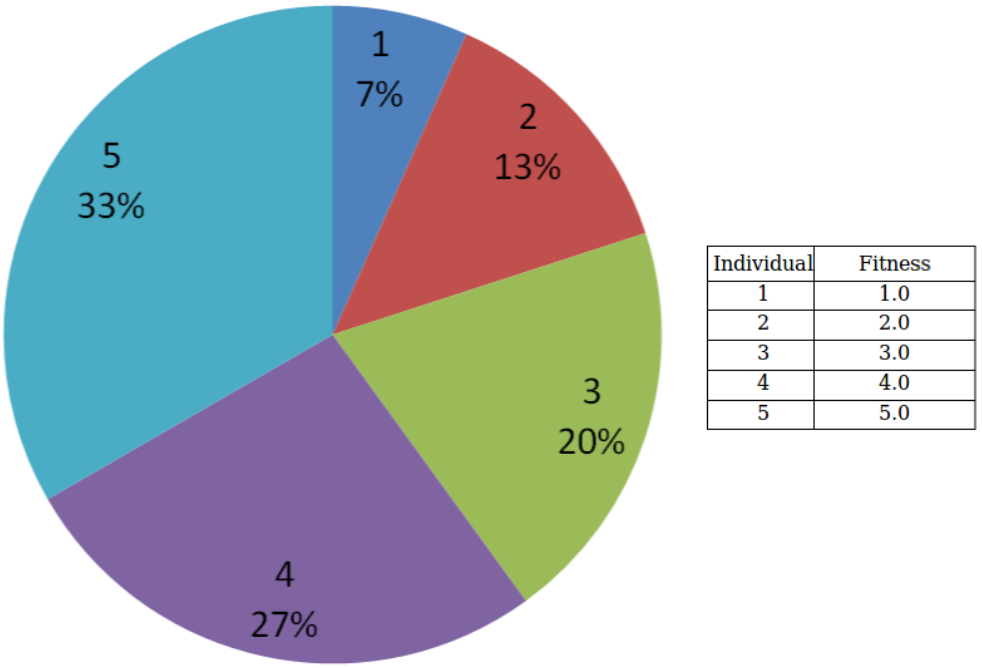
This selection types works differently. You can see in Code 3.2 that we have a tournament size of 4 and we chose to take 5 parents. Between all the results, we create a tournament which is a just a random list of AIPlayers. The size of the list is the amount equal to that of tournament size. Of those in the tournament, we choose the AIPlayers with the highest fitness. We repeat the same process for however many parents we need.

- Roulette Selection

```
@staticmethod
def roulette_selection(results, num_parents=5):
    total_fitness = sum(max(0.001, fit) for _, fit in results)
    selected = []
    for _ in range(num_parents):
        threshold = random.uniform(0, total_fitness)
        curr_sum = 0
        for player, fit in results:
            curr_sum += max(0.001, fit)
            if curr_sum >= threshold:
                selected.append((player, fit))
                break
    return selected
```

Code 3.3: Roulette Selection

This selection method, in Code 3.3, simulates how a real roulette wheel works in the real world. It is much easier to explain this selection visually, so take a look at the following photo:



Individual	Fitness
1	1.0
2	2.0
3	3.0
4	4.0
5	5.0

Figure 3.7: Roulette Selection Visual

As you can see in figure 3.7 [1], individuals with higher fitness are more likely to be chosen. It is given the name roulette wheel because you can also imagine it like a casino roulette wheel and the ball the dealer throws represents who gets chosen. In that example, the size of the hole the ball can go into depends on how much fitness the individual has; the more fitness they have, the bigger the hole the ball can go into which means the higher chance they will be selected.

To discuss the pros and cons of the methods, we must first discuss the idea of genetic diversity, convergence, and the exploration of solutions.

3.3.4 Genetic Diversity and Convergence:

An individual's genetics in my context, is the neural network's weights of the individuals. We would like for each generation at the start to try different approaches to the problem and that can only happen when there is genetic diversity in the population. If we have little to no genetic diversity at the start, the bots will have extremely similar outputs and they will get stuck without being able to learn the correct way to jump up from platform to platform. The best-case scenario is when the genetic diversity begins to drop as we approach the optimal solution to our problem. This dropping of the genetic diversity is called converging onto a solution. We want the bots to become more and more similar, the closer we are to the correct approach so they don't stray away from it. Therefore, both genetic diversity and convergence are important to consider, since we really want much higher genetic diversity near the start of the simulation to allow for genetically diverse bots to have different outputs for similar inputs.

This is called exploration, we want the bots to explore different possibilities and then hopefully converge closer to a correct solution when they begin to exhibit more and more similar actions.

This is why the different selection methods matter, since each selection method respects genetic diversity differently. Top-N selection drops genetic diversity quicker than the other two selections since it always chooses the top n performers, ignoring all other bots. However, this means that it could maybe converge on incorrect partial solutions which is an issue. Roulette is quite the opposite, it highly respects diversity, allowing for even very low fitness bots to be chosen, but still favoring higher fitness bots. This helps prevent early convergence but it could take much longer for it to converge also on a correct solution. Tournament selection is a midway point between

them, offering both a large advantage to higher fitness bots and still allowing for lower fitness individuals a chance to be chosen.

3.3.5 Initialization

In my case I have random initialization at the very start of the simulation. Both the weights and the biases of the neural network inside the agents' brains are randomized. They also have random colors assigned to them which serve as a visual aide to see which bot descended from who. Because from one generation to the next, the parent passes on its brain, as well as its color. I mutate the color slightly to show that the weights and biases were also mutated as well.

3.3.6 Neuroevolution

The idea of combining neural networks with evolutionary algorithms is called Neuroevolution. I would say that my program is the simpler form of Neuroevolution where the weights and biases are the only thing that I affect in the neural network. However, other implementations like the Neuroevolution of Augmented Topologies (NEAT) [9] show that not only can you change the weights in real time, but also the connections and structure of the neural network as well.

In typical evolutionary algorithms, the crossing over of genetic traits between parents to create the offspring is a vital step in the process. This is of course inspired by normal human biology. The parent's genetics would be represented by some sort of string or array and then mixed to create the genetics of the child.

In my case, the genetic material of the parents contains the weights of a neural network, which are interdependent. Therefore, you cannot simply mix and match the weights and expect a viable outcome with any real hope. The neural network's learning or knowledge is spread amongst itself and it breaks it if you randomly combine two different networks to one. Even though it is true in my case that implementing a

crossover step would ruin the learning process, other different implementations of neuroevolution like the NEAT method does include crossover. However, NEAT is structured in a way to take advantage of the crossover step instead of being harmed by it.

3.4 Testing

I have split up the testing into two major parts. The first part will include the testing of the actual written code of the game. Which is further split into white box and black box testing of the functionality of the various classes I have. The second part, due to the program being an experiment on evolutionary algorithms, will include some example runs I have personally conducted, which also function as manual UI checks. I will share my own thoughts and opinions about the results, but hopefully you try it for yourself as well!

3.4.1 Unit Tests

To test the code of the program, I will use unittest, which follows the xUnit testing style. The testing is further split into white box and black box unit tests. Black Box testing is done assuming no knowledge of the details of the code that you're testing. White Box testing is what tests the details and internals of the code.

Here is an example of both to understand better:

```
# White box
def test_brain_structure(self):
    layers = list(self.ai.brain)
    self.assertEqual(len(layers), 5)
    self.assertIsInstance(layers[0], torch.nn.Linear)
    self.assertEqual(layers[0].in_features, 8)
    self.assertEqual(layers[-1].out_features, 3)
```

Code 3.4: Test Example 1

Since this is a white box test in Code 3.4, we have internal knowledge of the code. When creating the ai's brain, there is a certain structure I expect to see. Therefore, I am checking if it truly is that structure I am expecting so I am not working off of any false assumptions. A black box test of the brain would not have knowledge of the inner workings of the brain, but instead it would test if it worked as expected.

```
# Black box
def test_decide_action_range(self):
    move_x,jump_dir,jump_strength = self.ai.decide_action(self.platforms)
    self.assertTrue(-1 <= move_x <= 1)
    self.assertTrue(-1 <= jump_dir <= 1)
    self.assertTrue(0 <= jump_strength <= 1)
```

Code 3.5: Test Example 2

In this black box test in Code 3.5, we have no access of the internal details of the function *decide_action*. We only care that the outputs are correct and usable based on what the code needs.

The Test Case ID showcases if it is a white box or black box test with a W or B respectively at the start of the ID.

With the explanation out the way, here are the rest of the tests:

- AIPlayer tests

Use Case	Test Case ID	Test Description	Expected Result
Agent's brain	W0-1	Testing the structure of the neural network that makes up the brain.	5 layers, made up of 3 nn.torch.Linear and 2 Relu, 8 inputs, and 3 outputs

Agent's brain	W0-2	Testing the mutate_weights function.	The weights of the brain to be different after using the function.
Agent's brain	W0-3	Testing the copy_weights function.	A copy of the weights to be the same as original.
Agent's color	W0-4	Testing colors of agents.	Making sure color logic is consistent.
Brain Inputs	B0-5	get_state ouput shape.	The get state output shape is what we expect the brain input to be.
Brain Outputs	B0-6	decide_action outputs	The outputs are indeed normalized to usable ranges.
Fitness	B0-7	Testing fitness with platform	A collision with a platform results in higher fitness

Table 1 - Testing AI Player

- Button Tests

Use Case	Test Case ID	Test Description	Expected Result
Button Properties	W1-1	Testing the creation of the button	Button has expected width, height, position, a text, and the object able to exist.
Checking Click	B1-2	Checking no click	No click should result in false.
Checking Click	B1-3	Checking simulated click	Simulate a window, a button, and a click over the button recognized.

Table 2 - Testing Button

- Controller Tests

Use Case	Test Case ID	Test Description	Expected Result
Top-N Selection	W2-1	Simulating a selection	We receive the highest fitness and correct number of parents.
Tournament Selection	W2-2	Simulating a selection	We receive the correct number of parents.

Roulette Selection	W2-3	Simulating a selection	We receive the correct number of parents.
Color	B2-4	Generating a random color.	We receive a random color that isn't too close to black (to avoid the players disappearing since screen background is black)
Color	B2-5	Mutating the random color.	Receive a different color from before.
Setup	B2-6	Casual Play Setup	Confirm all the objects are created and ready to go.
SetupAI	B2-7	AISimulation Setup	Confirm all the objects are created and ready to enter Simulation.

Table 3 - Testing Controller

- Drawing Tests

Use Case	Test Case ID	Test Description	Expected Result
Getting Information	W3-1	Getting information using get_info, killing one of the players, then again get_info.	Validate that get_info returned correct information both times.
Drawing Casual Play	B3-2	Attempting to do one draw	No errors, the function finishes without errors.
Drawing AI Simulation	B3-3	Attempting to do one draw	No errors, the function finishes without throwing errors.

Table 4 - Testing Drawing

- Menu Tests

Use Case	Test Case ID	Test Description	Expected Result
Base Menu Creation	W4-1	Create the menu	Menu is created with no errors
Main Menu Creation	W4-2	Create the menu	Menu is created with no errors
AI Simulation Menu Creation	W4-3	Create the menu	Menu is created with no errors

Instructions Menu Creation	W4-4	Create the menu	Menu is created with no errors
Simulation Summary Menu Creation	W4-5	Create the menu	Menu is created with no errors
Simulation Summary Graph	W4-6	Create the graph	Graph is generated with no errors

Table 5 - Testing Menus

- Platform Tests

Use Case	Test Case ID	Test Description	Expected Result
Platform Generation	W5-1	Generating Platforms	The correct number of platforms are generated.
First Platform	W5-2	Checking the first platform	First platform acts as the ground for the players to spawn on.
Last Platform	W5-3	Checking the last platform	Last platform is purple.

Platform Numbering	W5-4	Testing numbering	Platforms are numbered correctly, important since used for fitness.
Platform Spawn Location	B5-5	Bounds of Platforms	Platforms spawn within the width of the screen.

Table 6 - Testing Platform

- Player Tests

Use Case	Test Case ID	Test Description	Expected Result
Initial State	W6-1	Checking the Initial State of the player.	Players are in the middle, with no platforms reached, not charging, and they have control.
Jump Velocity	W6-2	Checking if jump increases velocity.	Simulates charging then releasing a jump, which results in more velocity.
Right Movement	W6-3	Checking movement to the right.	Player's x position should increase.

Left Movement	W6-4	Checking movement to the left.	Player's x position should decrease.
Grounded	W6-5	Checking if grounded, the player has control.	If on the ground, the player should have control.
Airborne	W6-6	Checking if airborne, the player does not have control.	If airborne, the player should not have control.
Starting Charge	B6-7	Trying to start charging a jump on the ground.	If on the ground, the player is able to start charging a jump.
Starting Charge	B6-8	Trying to start charging a jump while airborne.	If airborne, the player cannot start charging a jump.

Table 7 - Testing Player

How to run tests:

To run tests, you must navigate to the root folder of the project in the terminal. Remember, if you are using an environment to make sure the environment is active.

- To run all the tests at once, you run:

python -m unittest discover -s testing

- To run a singular test module, you run:

python -m unittest testing.NAME_OF_FILE

- Example:

python -m unittest testing.test_player

- To run a singular test class inside the test module, you run:

python -m unittest testing.NAME_OF_FILE.NAME_OF_CLASS

- However, I have structured all the modules to only contain one test class each for clarity and structure purposes.
- Example:

python -m unittest testing.test_player.TestPlayer

- To run a singular test inside a class inside the test module, you run:

python -m unittest
testing.NAME_OF_FILE.NAME_OF_CLASS.NAME_OF_TEST

- Example:

python -m unittest testing.test_player.TestPlayer.test_initial_state

Output:

Running all the tests at once should result in an output like this:

```
.....................................
----------------------------------------------------------------------
Ran 39 tests in 0.149s

OK
```

Figure 3.8: Unit tests Output

3.4.2 Results and Discussion

In this section, I would like to share some personal findings I have found out by running the simulation many times while developing the program. Other than the

specific settings for each run which I will state later, all the simulations were run using a mutation rate of 0.05.

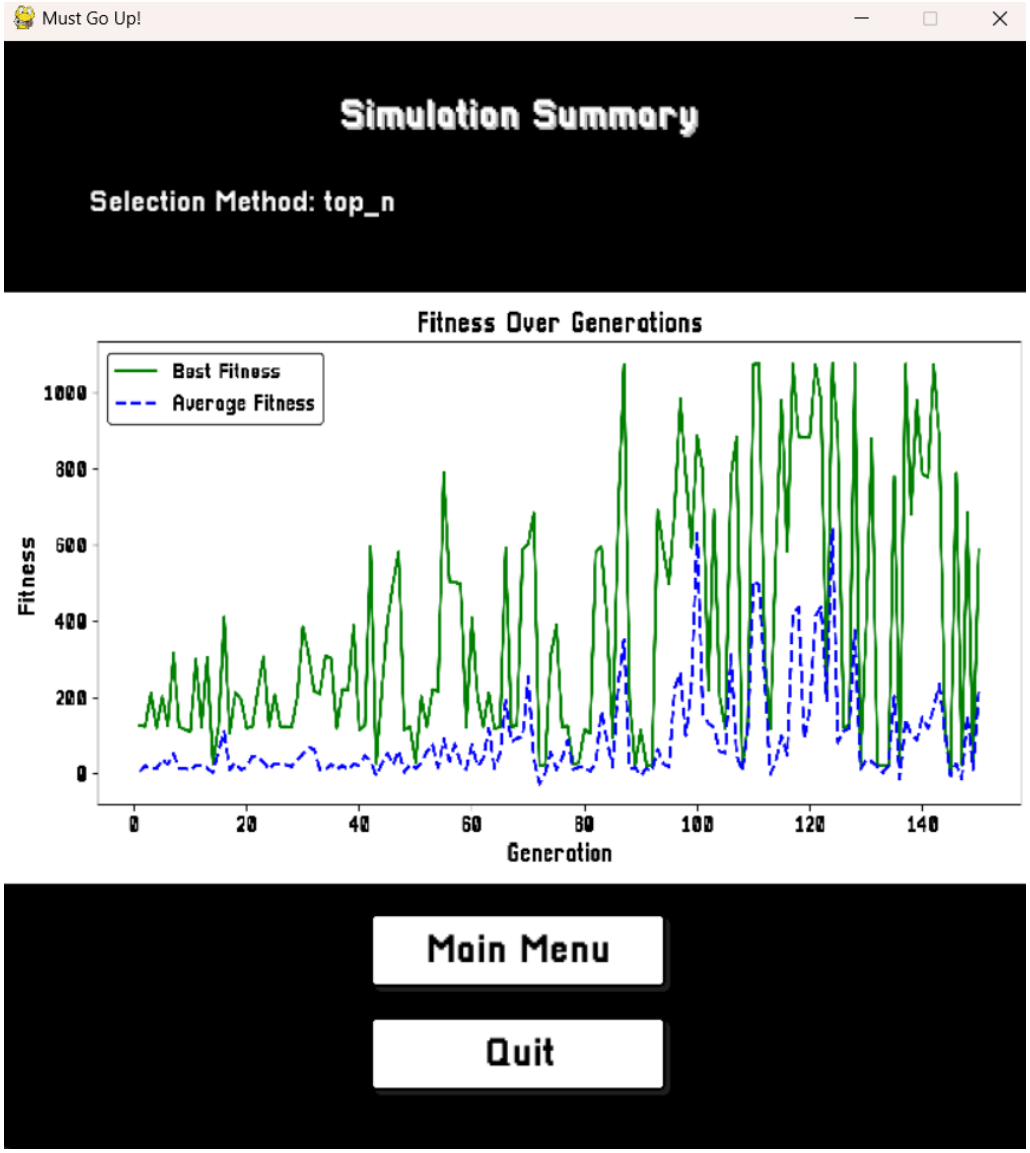


Figure 3.9: Simulation Top-N 1

Figure 3.9 shows an example of a simulation I ran with the default settings in the program. Which are a population size of 100, an eight second time limit for the bots to jump before moving on to the next generation, and using top-N selection. I ended the simulation at 150 generations. I would like to highlight some information in the graph.

- Firstly, it is very important to note that the graph is not consistently rising or falling. Like in nature, not every mutation will be beneficial to the population. After a good performance, you are not guaranteed that the next generation will perform better or even close to the last.
- Because the time limit is set at eight seconds, the maximum fitness the bots could achieve was around 1100, or jumping up 11 platforms.
- Before generation 60, you can see a large difference between the average fitness of a generation and the best performers. However, onwards, the difference becomes less and less which is a good sign that the entire population is at least heading towards the correct solution.

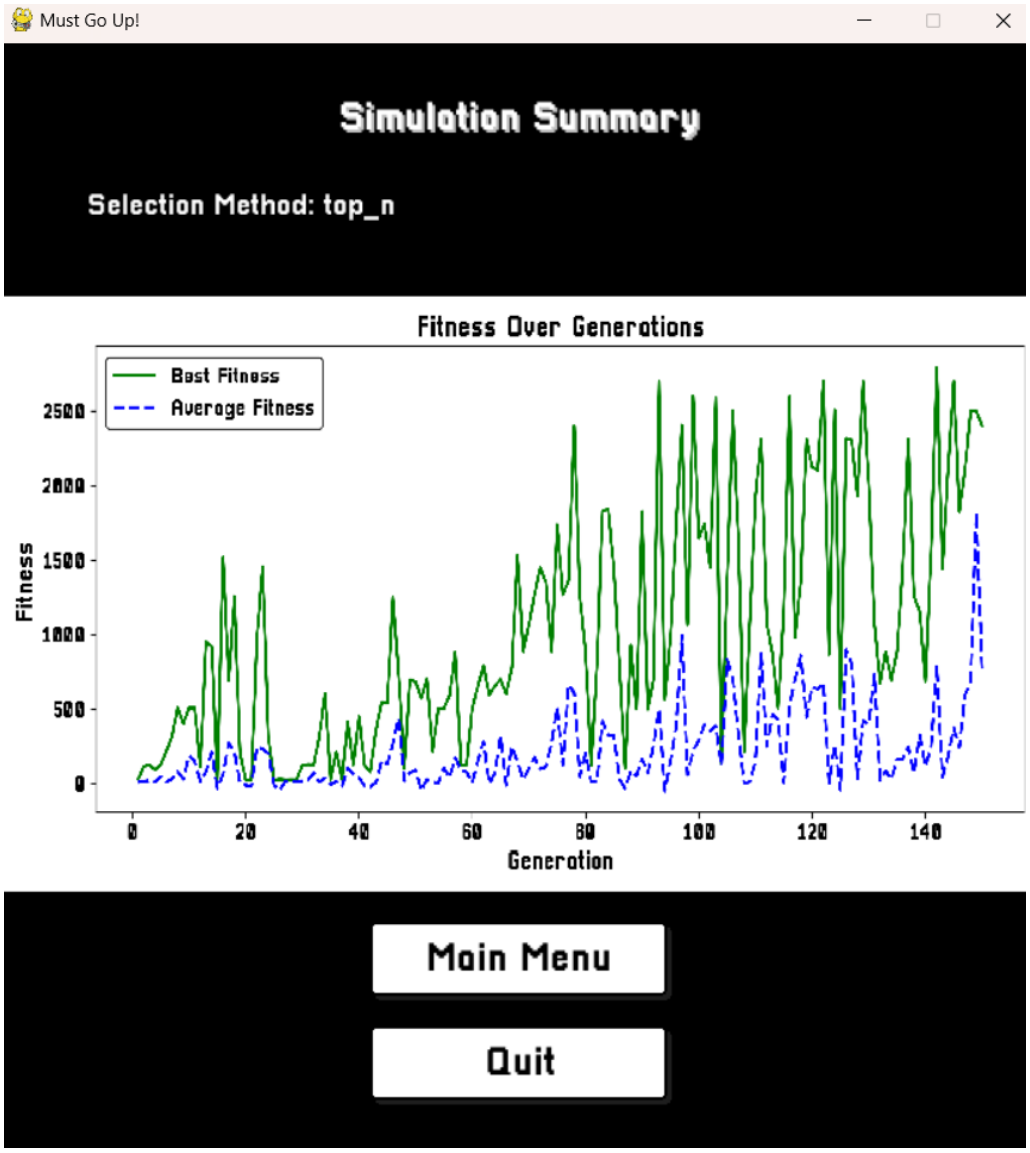


Figure 3.10: Simulation Top-N 2

This is a second example run, in figure 3.10, I would like to show. This was also done with a population of 100 and using top-N selection. However, I allowed the bots to run for a total of 20 seconds.

- I would like to note something interesting here that I have noticed happens sometimes. Around generation 20, the population achieves a relatively high fitness value; however, shortly afterwards it seems that the progress goes down quite low. I believe this is happening because the bots get lucky for a couple generations and learn incorrect behavior (for example always using maximum

height jumps). This issue is further magnified by the use of top-N selection, since the bots that learned the incorrect behavior are still chosen as the highest fitness and are the ones to spread their weights. It takes some generations to mutate away from this behavior back to correct learning.

- In this run, since the time limit was set at 20 seconds, you can see the bots were able to achieve a maximum fitness of over 2500 which means they jumped to above 25 platforms!

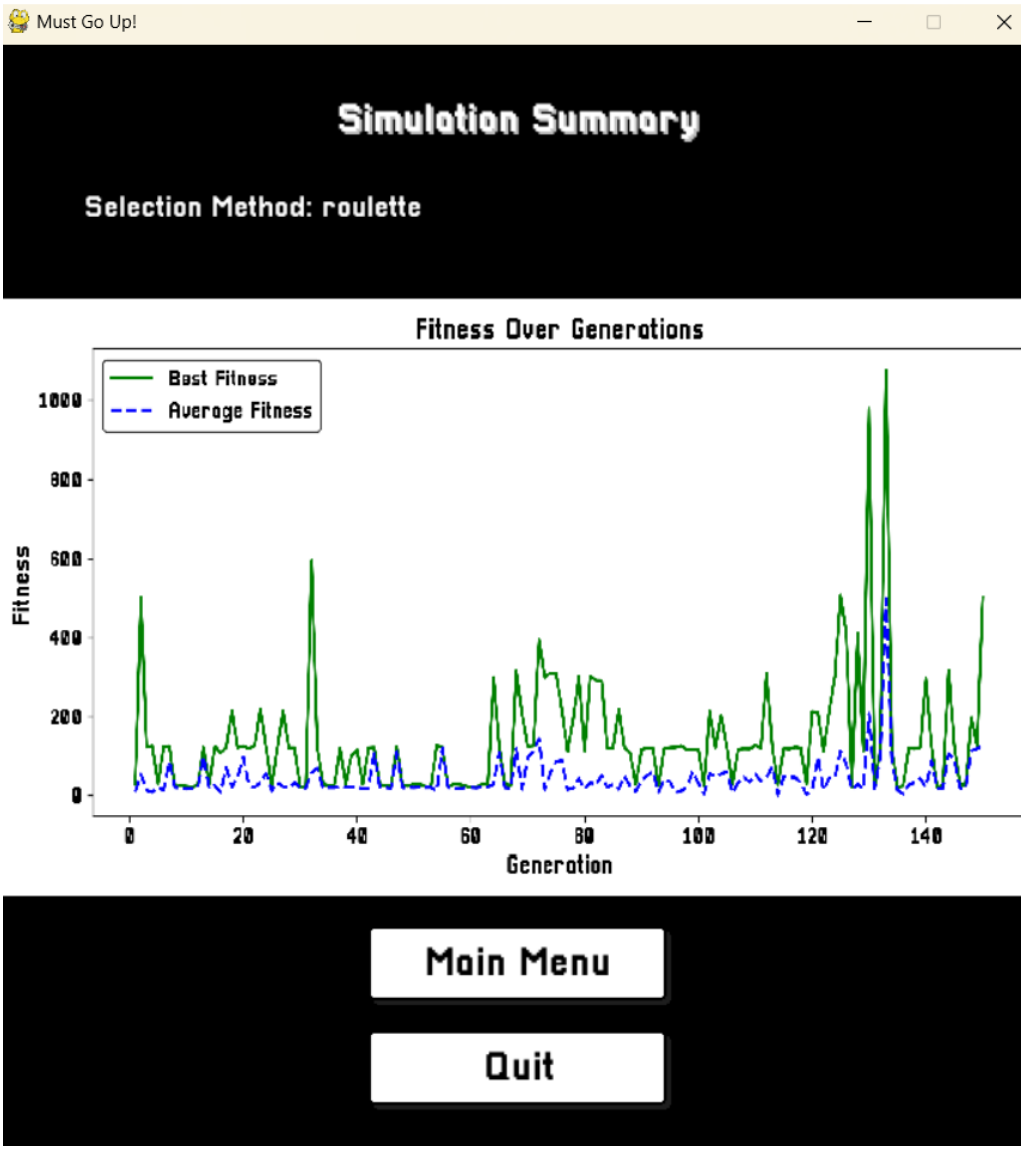


Figure 3.11 Simulation Roulette 1

Figure 3.11 showcases the results of a simulation I ran for the same 150 generations as before, with a time limit of 8 seconds and a population size of 100. However, in this simulation I chose to use Roulette selection. Roulette selection gives much less of an advantage to higher fitness individuals compared to Top-N selection, meaning we maintain way more genetic diversity. You can see the effect in the difference of their graphs. Even though we got some peaks around generation 30 and around generations 130 in this run, it fails to pass on the successful traits to the next generation. This pattern of very low fitness values and inconsistent peaks is what occurs again and again, you can see another example I ran for longer using roulette Selection in figure 3.12:

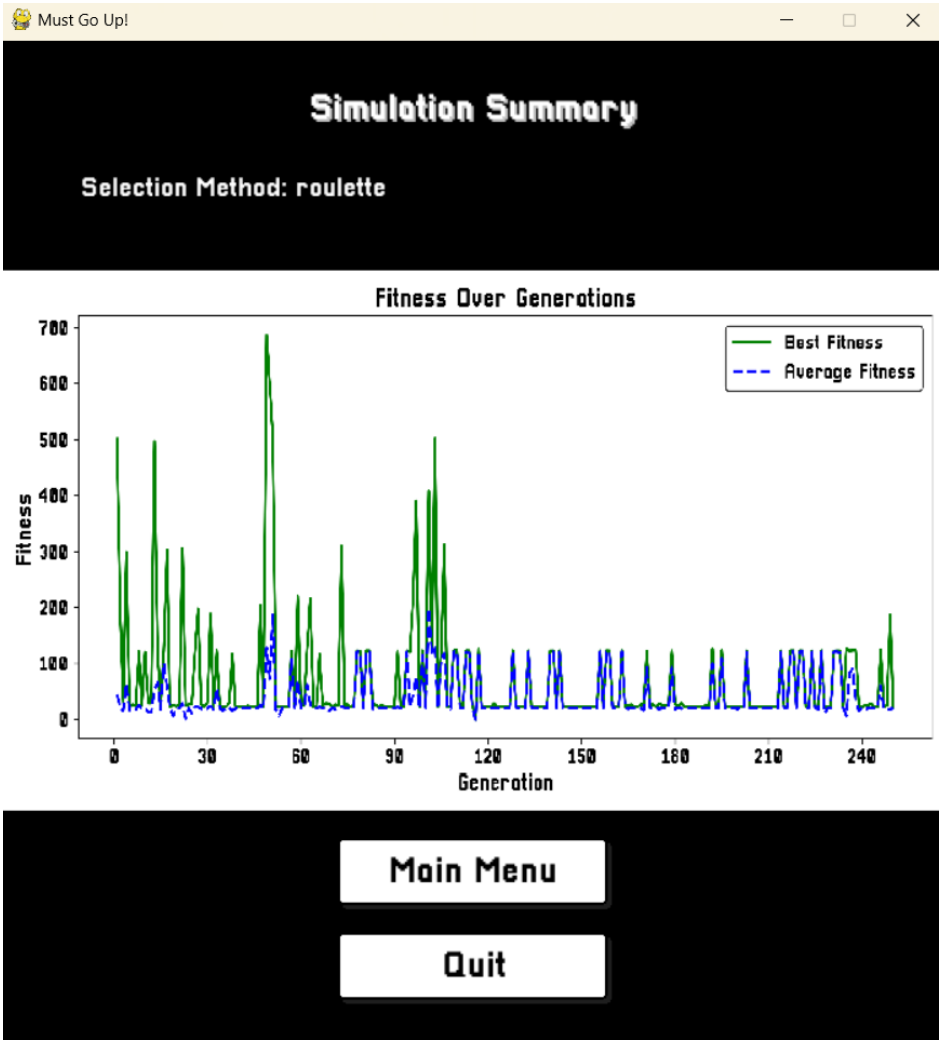


Figure 3.12: Simulation Roulette 2

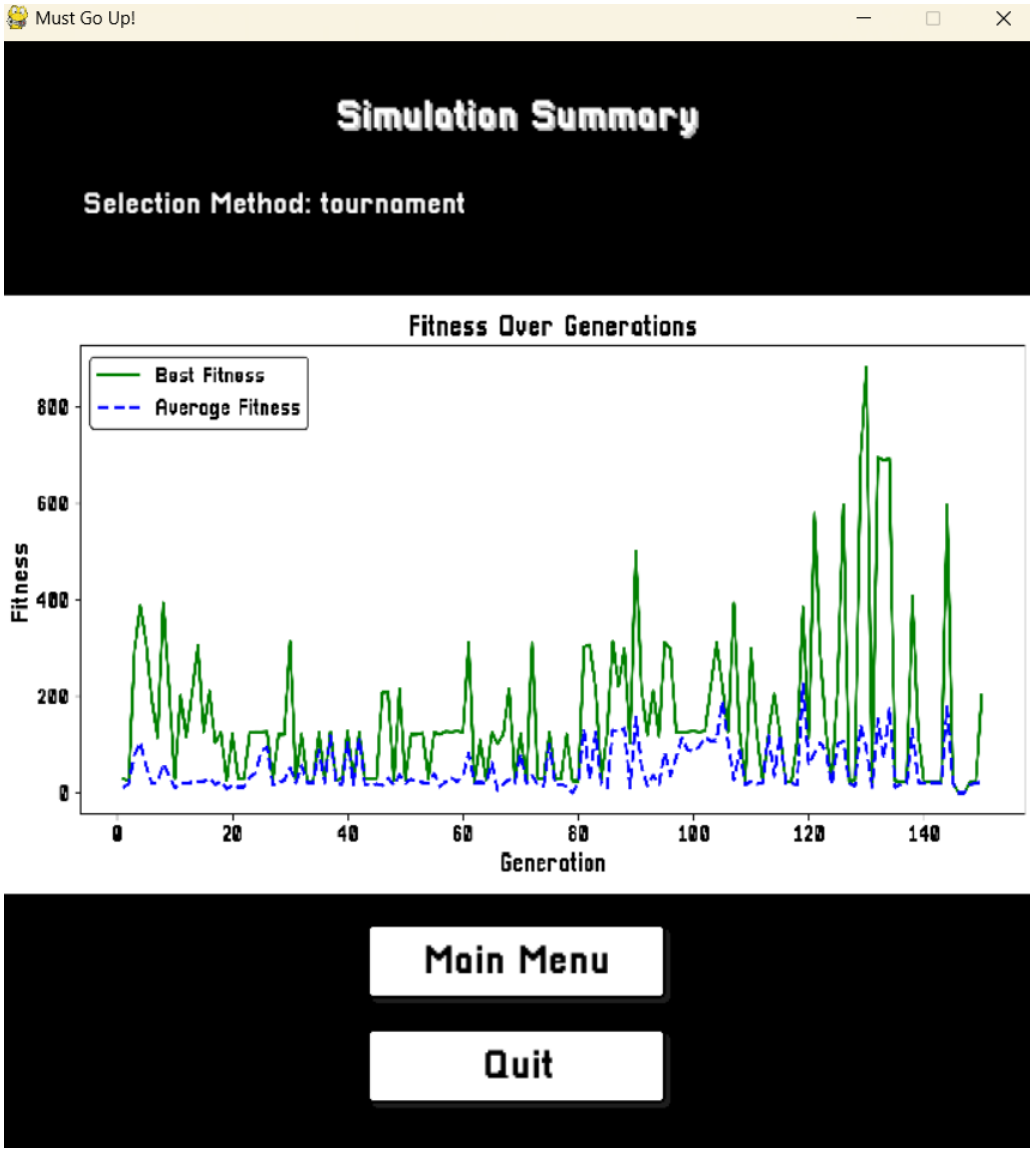


Figure 3.13 Simulation Tournament 1

Figure 3.13 showcases the results of a simulation I ran with the same settings as figure 3.11, but this time I use tournament selection. As discussed before, tournament selection maintains genetic diversity more than Top-N selection, but less than Roulette selection. We can see that in the graph, where there is a general upward trend a bit nearer the end around generation 130. However, since it does not completely favor high fitness individuals, you can still get heavy inconsistencies. You can see that in the longer run in figure 3.14:

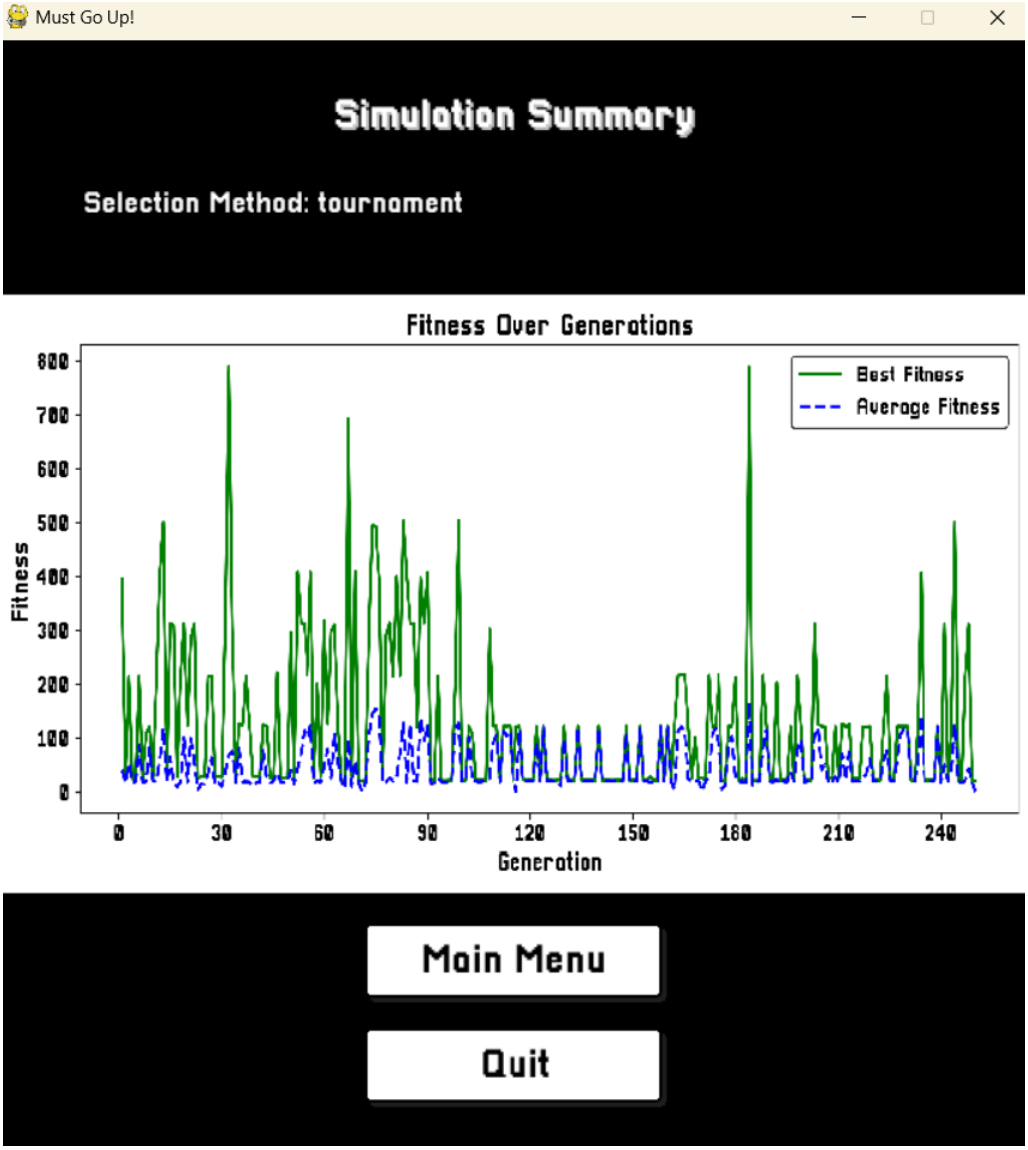


Figure 3.14: Simulation Tournament 2

- Summary

These findings give us information regarding the environment. The better performing bots are much better than those who are under performing. I say this because when using Top-N selection where the better performers are forcefully picked, the results are always the best. Tournament selection follows, where high fitness values can be semi-consistently reached, but it still is clearly worse than Top-N selection. Roulette comes at last place, where the higher fitness individuals have the least advantage among the three of them.

Since my environment seems to heavily favor high fitness individuals, it is to the population's disadvantage to not choose them very often. However, it is important to note that these results come from my own personal fitness calculations, if another person chooses to reward or punish the population differently, another selection method could potentially outperform Top-N selection in that case.

Chapter 4

Conclusion

4.1 Limitations

- The mutation rate is always set and constant throughout a simulation run.
- The time the bot lives is also set and does not change. This means that even if some bots are currently still jumping and achieving higher fitness values, the simulation will still end the generation and move on to the next.
- Having to access the source code to change the details of the simulation.
- Platform placement is always random.
- The only obstacle currently is having to jump up from platform to platform.
- Due to the large number of bots, the camera always will follow the highest position bot, but this can be visually irritating sometimes.
- I personally was not able to find the absolute "best" mutation rate, only a range of good values.

4.2 Future Work

- The addition of an option for mutation rate to decrease throughout the simulation, to converge on an "optimal" solution.
- A settings menu for various options, like difficulty of environment, simulation settings like population size, mutation rate, number of platforms, and more.
- The addition of a feature to allow for an individual performing well (still increasing his fitness value) to extend the time limit of a generation.

- Different difficulties, a set platform level, easier platform placements, more difficult ones, and so on.
- Introduce new obstacles, like enemies, walls, spikes, and so on.
- Implementing a better camera or way to track the bots in the middle of the simulation.

4.3 Conclusion

The architecture of the program is built using the model view control pattern, which separates a program into data and logic, drawing and user interface, and the controller which links them together.

After a random initialization of a population, bots' actions and how they perform in the environment determines each of their personal fitness values. The fitness values are used in the different selection types the user is able to click on from the user interface. Finally, at the end of a run, a graph showcasing the summary of the current simulation is displayed.

My thesis explored the idea of implementing an evolutionary algorithm in a two-dimensional platformer. The AI agents are able to receive inputs, process them with their "brains" which is a neural network and decide their next movement based on them. Through the process of allowing them to play the game, giving them a fitness score, then selecting the parents for the repopulation of the next generation, I was able to make the bots learn how to play the game.

Acknowledgements

I would like to thank my supervisor, Mr. A. H. M. Sajedul Hoque for helping and guiding me throughout the process of writing my thesis. I would like to extend thanks as well to the entire university, and to all the professors that have taught me since the first semester. It was a great honor to be taught and instructed by everyone at the faculty and it was a journey I hold dear to me and one I will never forget or take for granted.

Bibliography

[1] Amorim, Elisa & Xavier, Carolina & Campos, Ricardo & Santos, Rodrigo. Comparison between Genetic Algorithms and Differential Evolution for Solving the History Matching Problem. Lecture Notes in Computer Science. 7333. 635-648. 10.1007/978-3-642-31125-3_48. (2012).

[2] Darwin C. The Origin of Species: By Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life. 6th ed. Cambridge University Press; https://doi.org/10.1017/CBO9780511694295, 2009.

[3] Mehrdad Dianati, Insop Song, Mark Treiber, *An Introduction to Genetic Algorithms and Evolution Strategies*, Ontario, Canada, University of Waterloo, 2018.

[4] Ian Goodfellow and Yoshua Bengio and Aaron Courville, Deep Learning, MIT Press, *https://www.deeplearningbook.org/*, 2016.

[5] MatPlotLib, *https://matplotlib.org/*, Retrieved April 28th, 2025.

[6] Python Documentation, *https://docs.python.org/3.13/*, Retrieved April 28th, 2025.

[7] Pygame Documentation, *https://www.pygame.org/docs/*, Retrieved April 28th, 2025.

[8] Pytorch, https://pytorch.org/, Retrieved April 28th, 2025.

[9] Kenneth O. Stanley, Risto Miikkulainen; Evolving Neural Networks through Augmenting Topologies. Evol Comput 2002; 10 (2): 99–127. doi: https://doi.org/10.1162/106365602320169811, 2002.

[10] unittest, https://docs.python.org/3/library/unittest.html, Retrieved April 28th, 2025.

List of Figures

User Documentation:

Developer Documentation:

List of Tables

List of Codes