Final Project Writeup

Connor Holm
University of Minnesota- Twin Cities
Minneapolis, Minnesota, USA
holm0850@umn.edu

Jonathan Blixt
University of Minnesota- Twin Cities
Minneapolis, Minnesota, USA
blixt013@umn.edu

1 Abstract

Genetic algorithms are a subclass within artificial intelligence that allows for various problems to be solved. Dating back to the 60s, genetic algorithms have helped solve many solutions. The concept behind genetic algorithms referred to Darwin's ideas on natural selection and evolution. As time passed, the applications of genetic algorithms spread into many different industries. In this research paper, the process of solving a video game through the use of a genetic algorithm was built. Flappy Bird was a popular mobile game that was launched in 2013. The goal was to create an agent that could successfully play the game. The format and methods for developing a genetic algorithm have been researched, however, implementing it into an existing code base for a specific game is a new challenge. By the end, an agent could be created that had optimized parameters through the various processes of a genetic algorithm.

2 Problem Description

Initially, the game of Flappy Bird seems quite an easy tap to jump and survive, however, the precision and coordination of jumps are critical to success leaving many players frustrated and wanting to find a better way to play. The game itself is a simple 8-bit graphics style in which the agent is a bird centered horizontally on the screen where the background moves by with obstacles. In-game the obstacles are as follows the top and bottom of the screen, which are represented by the ground and sky. However, the challenging obstacles in the form of pairs of pipes that are created in even intervals horizontally with a fixed size gap at different heights on the screen. The challenge lies in chaining together jumps to not only pass through the current pipe, while prepping to pass through the next pair as well.

2.1 Why Flappy Bird?

This would be an interesting project to work on as it would be a fun way to beat a game that was a large part of society at one point. Additionally, this app was known for being difficult to get a high score on, so it would be an engaging task to create an algorithm that can score well in a traditionally difficult environment for humans.

2.2 Game Challenges

2.2.1 Game Interface. The original game is a mobile game making sensors and actuators for any agent to play the game needlessly complicated. The agent would first need a screen capture of the screen itself to know what is happening, and even that would still need some vision processing algorithms on top of it to determine useful data as discussed in the section Learning Optimization Parameters. For sensors, a simulated input would be needed which on mobile devices is quite difficult with current limitations of apps to simulate inputs over others. So, the mobile game is a poor choice of an agent so a Python port of the game in which both sensors and actuators can be simplified by directly assessing data assets and inputs to the game make this task feasible.

2.2.2 Dimensionality. However, even given access to the birds' exact position and pipe position the agent still needs many different dimensions of sensor data like height from the ground, x distance to next pipe, y distance to top pipe, y distance to bottom pipe, and more. The high dimensionality of this data may require either some form of dimensionality reduction or simply more computational power. For example, if the height from the ground needs to stay at an optimal height then the agent needs to not only determine which actuators can achieve that optimum, but must also consider other parameters as well in the optimization process.

2.2.3 Environment. Given that the game is partially observable, in that pipes are not all visible to the agent, and that it has an infinite number of game states makes graph traversal search algorithms like A* are impractical for this kind of application. The infinite procedural generation of pipes and randomness of the game makes the environment non-deterministic, forcing the agent to be more complex. Because of this, the agent will need to plan ahead and take jumps that will not only get through the current pipe, but future pipes as well requiring the algorithm to track these features.

2.2.4 General AI Challenges. However, with any problem, especially a genetic AI application, overfitting must be avoided to make the agent as responsive as possible. If overfitting occurs the overhead of the algorithm could cause the agent to perform poorly by having input latency issues.

2.3 Software

The software that we will be using for this project includes an open-source repository that already has a working Flappy Bird game [1]. This repository builds the game in Python using the pygame library. For coding a genetic algorithm, we will also be using native Python.

2.4 Preliminary Work

We haven't done any preliminary work on this project ourselves. As mentioned above, we will be using an open-source repository that already has a working Flappy Bird game allowing us to focus on the genetic algorithm.

3 Background

3.1 Background Introduction

Genetic algorithms have had a large impact on the way algorithms have been developed. The origin of these algorithms came from the ideas of natural selection in genetics. Hence, the name genetic algorithms. The process for creating a genetic algorithm at its core aligns with the idea of the fittest member of a population surviving and being able to reproduce. This leads to a well-performing model after many generations. As the use of this algorithm came to be, it expanded into many different industry sectors which are discussed later in the paper.

3.2 History Behind Genetic Algorithms

One of the first instances of this was in cybernetics movement applications in the early 60s [3]. The idea behind the algorithms used was based on natural phenomena and was used because these systems were mostly emulating "natural systems." At the time of their inception, Genetic Algorithms weren't particularly successful or used until other technologies developed like neural networks. The creation of these algorithms and methodologies are often very intertwined with one another even if alone, they often perform relatively different tasks

3.3 Early Genetic Algorithm Uses

Some early usages of genetic algorithms that are still common today are in classifier systems. The classification algorithms take sets of known samples and "learn" their attributes through given classifications generating probabilistic models. The models can then be used to classify previously unseen samples by finding similarities in subsections of the total data sample. These new samples can also be used to improve the accuracy of the algorithm over time.

3.4 Naming Rationale

The terminology itself comes from organisms and different evolutionary theories in which multiple "parents" produce "children" with a combination of characteristics. The naming of "Genetic Algorithms" may be a bit of a misnomer because often many "genetic" systems that fall under that heading have parallelism, which contrasts typical natural evolutionary systems where the old must die and the new ones are born [3]. The slow speeds of natural evolution are also very common criticisms for the naming scheme, but when new populations and generations can be created many times per second and larger amounts of "mutation" can happen per generation, these concerns are quickly ignored. So, overall the genetic algorithm may not be the most precise name, but for the sake of simplicity, we use it anyway.

3.5 Current Applications

Genetic algorithms have been implemented to solve challenging problems. These areas of applications include categories within operation management, multimedia, and wireless networking [5]. The genetic algorithms provide a solution that is either effective or uses fewer resources than other more complex algorithms.

3.6 Operation Management

According to Katochl, facilities that were clustered used mutation and heuristic operators for a genetic algorithm were 7.2% less than other algorithms [5]. Additionally, there have been genetic algorithms implemented that can help manage inventory control. The algorithm also helped provide information on how many warehouses were needed to store the data. Finally, genetic algorithms have been implemented in financial trading institutions to help make predictions. Something that is important to note is that these genetic algorithms have also been combined with other models like neural networks.

3.7 Medical Technologies

Genetic algorithms have had applications in both image and video processing [5]. It requires a lot of computational power to partition an image yet genetic algorithms do a better job as their search capabilities are effective. There are many actions that genetic algorithms can take on an image or video. Some of these include; improving the contrast, merging the noise and color attributes, and magnifying a picture. Additionally, genetic algorithms have been known to be able to help with detection in CT scans in the medical field [4].

3.8 Other Applications

It has been implemented in a way where the genetic algorithm is combined with a neural network to help identify brain tumors [4]. Additionally, genetic algorithms have been used in agriculture for finding the capacity of water in the soil, using remote sensors. A large part of media is games. Genetic algorithms have been able to solve many different games. One of these games was Gomoku [13]. It performed better than other tree-based algorithms.

3.9 Wireless Networking

Genetic algorithms were used to help allocate bandwidth for various channels. Additionally, they were used in wireless networking to help find wireless nodes (localization). The algorithm used simulated annealing along with a genetic algorithm to find the position [14]. It was interesting that the average calculation time was 50% faster when simulated annealing was added to the genetic algorithm. As stated

in the journal, the model was built using the various steps defined in this paper's Creating a Genetic Algorithm section.

3.10 Creating a Genetic Algorithm

The process of creating a genetic algorithm through various steps [11]. The first step involves creating an initial population. Then evaluating the fitness of each population member. This step can be done through something like a cost function. After evaluation, you need to start natural selection. This means only letting the best members of the population live. The next steps involve mating. First, the members need to be paired for mating. Then an offspring needs to be generated from the parents. Additionally, the offspring can be selected for mutation. This process can be continued or ended depending on how the population is performing. The steps for a genetic algorithm are better described below.

3.11 Creating a Population

When creating a population, there are many different methods but it usually involves creating many samples through a random function. These can be stored in an array or even a matrix. These samples should have enough variation that they are able to evolve throughout the run-time of the algorithm.

3.12 Evaluating each Member

The evaluation of each member is the key determinant of what is good vs what is bad. In order to effectively evaluate a member of a population there needs to be a cost function. This cost function will return the fitness level of a member. An example of a cost function could be,

$$\text{cost}(x) = \sum_{i=1}^{N} x_i^2$$

[11]. This would take the square of all the attributes in a member and sum them together. However, there are many different types of cost functions and certain functions will be better for specific data sets.

3.13 Natural Selection

The process of natural selection involves selecting a k amount of members from a population that have the lowest cost. This idea states back to Darwin's Ideas on natural selection where only the fittest members of a population will be able to survive. Eventually, the process of only keeping the best members creates a population that performs well.

3.14 Mating

Mating is the idea of pairing members of the population together for breeding. The fittest members of the population are the ones that are most likely going to be chosen for reproduction. To simulate this process, a typical way is through a roulette wheel [11]. This involves giving the fittest members a higher probability of being selected for mating than weaker members. The percentages that describe a member's chance of getting chosen will need to be tinkered with. However, once the probability is set, it doesn't need to be changed as the number of members that survive natural selection will be constant throughout the evolution of the population.

3.15 Create offspring

When creating an offspring, it should be a combination of both the father and mother. There are many different ways to combine attributes from both parents. However, one possible way is through the use of bit masks. This process can be done by creating a random mask of both the mother and father. These two masks can be applied in various ways to create many offspring with parts from both the mother and father. The most common practice for creating an offspring involves the process called crossover. This allows the parents to swap bits with each other at a point or multiple points in the bit string rather than a [7]. This gives the offspring a chance to inherit genes from both parents. Notice, that there are many different parameters that need to be looked at. These include the rate at which crossovers are implemented, and how many crossovers are used in the process of creating an offspring [7].

3.16 Mutate offspring

In genetics, mutations can arise in creating offspring. This change in the DNA is not from either parent but rather a random change in DNA. Mutations from genetics can also be applied to genetic algorithms. This process allows random change to the offspring. A common practice for this method is called bit-flip mutation [7]. It involves switching a bit value from 0 to 1 or 1 to 0. When it is not bits, it can simply be switching whatever is representing a gene into a random or opposite value. Additionally, mutations should be occurring at a lower rate than crossovers. If this is not the case, then it becomes more of a random evolution rather than genes being passed down.

4 Genetic Algorithms in Games

Genetic algorithms can be a great way to find an optimal solution for a simple game. There are many different things to consider when developing an algorithm to help solve a game. For one, the parameters of one game will be much different than another game. This means that an effective genetic algorithm that works with Gomaku may not work with some things like Chess. Another application that genetic algorithms have been used for is scene generation [8]. This is a key part of video games to simulate endless levels.

4.1 Scene Generation

In order to generate a scene in a video game, there are many considerations to be taken. The scene may need to include

a set of walls, enemies, collectibles, a starting point, and an ending point [8]. A key realization is that these scene parameters can be represented as a matrix. The digits in an element in the matrix can represent one of the parameters. Depending on the game it can be a 2D or 3D matrix to define the scene. Once the cost function has been effectively set up then the algorithm can start to optimize the generated scenes. Once working properly, it can give the illusion of an endless world if scenes can be infinitely generated.

4.2 Card Battle

Genetic algorithms can also be used to effectively play as an agent in games that are referred to as card battle [9]. The difference between a card game and a card battle resides in the number of attributes that are being analyzed by an agent. In a card game, only one attribute is analyzed while a card battle has many attributes being analyzed. Optimal algorithms have already been developed for card games, however, these same algorithms wouldn't be considered optimal when there is more than one attribute. A few of the considerations that go into a Card Battle include the cards in your hand, the position of the cards in your hand, the cards in the enemy's hand, and the position of the cards in the enemy's hand [9]. All of these attributes need to add to the cost function so they all can be evaluated in some way. By the end, the genetic algorithm will help effectively play the Card Battle in a better way than just looking at one attribute.

4.3 Case Injected - Genetic Algorithms

Case-Based members of a population have predetermined moves and have performed well in previous games or populations [6]. This allows for an improved genetic algorithm in both quality and time. This improvement comes from occasionally adding good members that have good performance in other problems. The idea behind case injecting is that it can combine old game scenarios into a new method to make it perform better. In one case, implementing a case-injected genetic algorithm allowed the agent to perform with 5% of the optimal target 95% of the time [6]. Keeping track of the cases that perform well can be done at the natural selection step during the evaluation of the cost function. Based on the desired goal, the number of cases that can be kept for injection can be varied. However, in the example used for a real-strategy game, 15% of the top cases were used for injection. Though one may find the natural selection step and the idea of keeping the top 15% cases very similar, however, there is a key distinction. This distinction lies in the idea that natural selection only keeps a certain amount of members in the current generation population. Yet, the cases used for injection are the best members from all generations of the population.

4.4 The Endless Runner Genre

Since Flappy Bird is in the Arcade Genre and more specifically the modern Endless Runner subgenre. These games have very limited environments that are infinitely generated in a pseudorandom way in each play-through [10]. The levels often times increasing in difficulty over time by speeding up the camera requiring more precise user input over time. These games have become extremely popular as mobile games in the past decade. They all have a very simple goal survive as long as possible by running, flying, driving, or whatever medium the game has chosen. The simplistic platforms often only have a few potential user inputs like turning by swiping or simply tapping the screen to jump, making the timing of user inputs much more critical than any specific inputs. This means agents have very few potential actions as well. Overall with few sensors and actuators, the internal learning model is reasonable in complexity to be used in online environments and perform in real-time.

4.5 CoinTex

CoinTex is an arcade-style game in which agents need to collect coins while avoiding obstacles such as fire [2]. The agents than can then collect data on where coins and obstacles may be in the environment. The genetic algorithms fitness function simply rewards an agent moving near coins and punishes movements that collide with fire or other obstacles. The agent then takes a given weight, which to start is randomly generated and applies its given precepts to generate an action. The agent then outputs its fitness score after this move is taken. Many agents are run in parallel until they reach a goal state, either collision with an obstacle or having collected all coins. The best are chosen by taking the summation of fitness function after all steps to create a new "generation" of agents to then repeat this process, ideally with higher fitness scores over time. After a given number of iterations, the program can terminate and output its "ideal" weights to play the game. The ideal weights are the "genes" or instructions that the agent should use to make decisions

4.6 Neuroevolution of Augmenting Topologies (NEAT)

The NEAT algorithm is a specific variant of a genetic algorithm implantation, in the most basic sense, utilizing a dynamic neural network by changing the internal structure of the graph by making more or fewer connections based on the complexity of the environment, hence the augmented topology [12]. This approach, compared to traditional genetic algorithms using static neural networks, is more likely to find efficient and robust solutions. The reasoning is because often times agents have extremely high dimensional input through precepts and many can be quite inconsequential in action decisions, so by adding layers to the neural network dynamically the algorithm can adapt to high dimensional

data quickly, making new generations of improvement rates higher and requiring fewer total iterations. The allowance for more layers also makes it so that solutions can become more complex given enough iterations and optimization. The increasing complexity over time without just needlessly starting with more layers than needed is a very attractive behavior preventing overfitting or degraded performance, by having an overly complex model. The NEAT algorithm has allowed agents to become increasingly complex given enough time

4.7 Alternative Methods for Offspring Creation

As discussed in the section Creating a Genetic Algorithm, crossover and mutation are typical methods for producing an offspring. However, there are other methods that can also be implemented. One of these methods is called inversion. This is the idea of reversing a part of a set of bits in the original string. This is shown below.

$$1010\mathbf{0101}01 \rightarrow 1010\mathbf{1010}01$$

Similar to a mutation in the sense that it can be applied randomly. This has the ability to add a different form of randomization to the offspring.

5 Approach

Creating a genetic algorithm varies based on the problems that are being solved. In this project, the goal was to get an agent to play Flappy Bird able to achieve a reasonable score. In order to get an agent to perform well, it needed to be optimized through many generations. The first thing was to figure out what a high-performing agent needed to be sensing when playing the game.

5.1 Learning Optimization Parameters

When playing the game, the agent must make it through the two pipes in order to increase the agent's score. This led to the two main things that needed to be observed; when to start tracking the next pipe and how far above the lower pipe to jump. This is shown in Figure 1. Once these different parameters where optimized through the genetic algorithm, the agent will have enough knowledge to navigate its way through the current and upcoming pipes with ease. These two parameters seemed to be the most effective given the amount of computing resources need to make the game run with a large number of agents. These two parameters would allow the game to run smoothly as it didn't have to calculate a large dimension of parameters for a large number of agents simultaneously. Naturally, the parameters will start off as random but will slowly lean towards an effective set of parameters as the weaker agents aren't able to pass their genes onto the next generation.

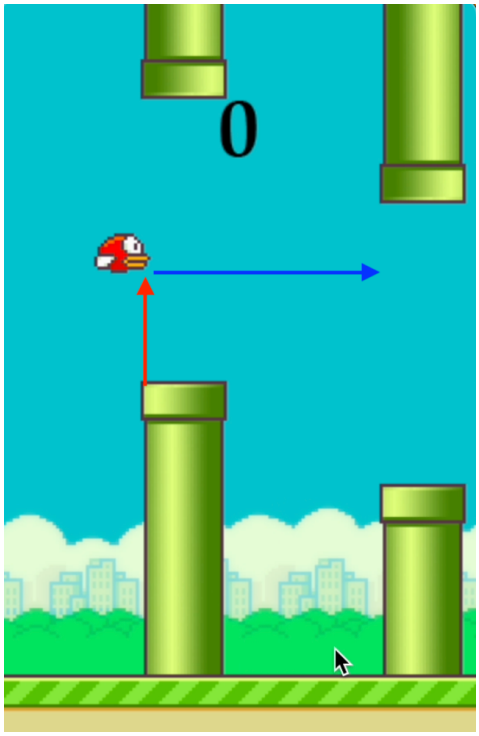

Figure 1. The red arrow suggests the optimization parameter for the height to flap above the lower pipe, and the blue arrow suggests the optimization parameter for when to start tracking the next pipe.

5.2 Developing a Population

The next part of the approach was to build a population that could run multiple agents at the same time. This follows the idea that a genetic algorithm has a population with multiple members all fighting to pass their genes onto the next generation of offspring. Originally, this game is only made for one agent at a time. This meant that the source code needed to be modified to account for multiple agents running simultaneously. This meant that there needed to be variables that track multiple birds' velocity, position, and score. These things all needed to be updated effectively so that the game would be able to run smoothly. After the changes to the game's source code took place, a population with varying sizes could be run seamlessly. A population size of 20 seemed like a good number to run the project with as it was enough to get significant changes in the agent's parameters between generations, yet it wasn't too many agents that the game couldn't run smoothly. With the population size set to 20 agents, the next part of building the genetic algorithm was to make sure that the agents' fitness was properly evaluated.

5.3 Evaluating Fitness

Evaluating an agent's fitness can be a hard process in some scenarios. However, for score-based video games, it is not

difficult. After each generation, an agent's fitness could be determined by looking at the agent's score which in Flappy Bird is a direct representation of how many pipes the bird made it through. Additionally, the agent's fitness is used in the selection process of natural selection.

5.4 Applying Natural Selection

Knowing the fitness of each of the 20 different agents, the natural selection process can be used to pass the well-performing agent's genes onto the next generation. The natural selection process used in this genetic algorithm involved looking at the top 5 agents and using their genes to be passed on to the next generation. This process simulated Darwin's ideas that only the fittest members of a population are able to survive to pass their genetics onward.

5.5 Mating and Creating Offspring

Mating can be a complicated process for genetic algorithms and can vary a lot between different problems. Some common procedures for building an offspring from two parents can include some process of crossover or mutation with bit-strings. However, the parameters that the agents in Flappy Bird game had were not able to be mutated or put through the process of crossover. However, it is still possible to randomly select new trait values with influence from both parents. In order to create a new offspring, two of the five parents from natural selection are randomly paired together. The next step was to create the offspring using the two parents. The process of creating offspring parameter values involved selecting a random value from a Gaussian distribution with the mean being the mean of the parents to values and the standard deviation being the absolute value of parent1's value minus parent2's value divided by two. This is shown in Equations 1 and 2

$$\mu = \frac{parent1 + parent2}{2} \tag{1}$$

$$\sigma = \frac{|parent1 - parent2|}{2} \tag{2}$$

This leads to a random value being selected that is likely near or between both parents. This process is repeated for both parameters being optimized. Additionally, the entire process is repeated 20 times to create a new population that is the same size as the old one with updated agent parameters.

5.6 Implementing New Generation

Once this new generation has been put through mating, the final step is to implement it back into the game as the new generation. This process of natural selection and mating repeats itself enough generations have passed where the agents are performing at a high level. This programmatically simulates Darwin's theory on natural selection and builds a well-performing agent in the process. It is important to note that each generation's parameters are recorded for analysis after the game has been completed.

6 Experimental Design

This project's goal was to create a Flappy Bird agent that was able to effectively play the game through the use of a genetic algorithm. The source code for the game was already written and it was important to redesign the game to support multiple agents to play at the same time. The design process for this project followed the basic outline described in the approach section of this paper. This meant that values like the population size, the number of members chosen by natural selection, and the number of generations before the agent could be considered high performing needed to be chosen accurately. In order to implement well-running code, these specific constants needed to be chosen effectively.

6.1 Modifying the Source Code

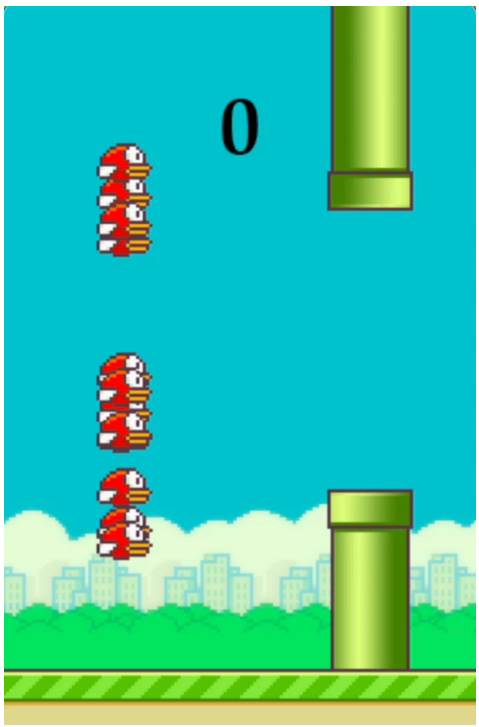

Figure 2. The game is played with multiple agents instead of just one agent.

The Flappy Bird source code was only designed to account for one agent at a time. This means that every variable that was related to the actions of the original bird needed to be modified so that it could handle an unspecified length of an array of birds. Specifically, this meant accounting for the velocity and location of multiple birds rather than just one. This may seem relatively basic, however, it becomes a computational challenge when the location of each agent in

relation to both of its optimization parameters needs to be tracked simultaneously. This code was added into the main while loop of the game and it is able to run smoothly behind the scenes. This is shown in Figure 2. Once the source code for the game was modified to allow for multiple agents, the next step was to find the important values of the genetic algorithm's various steps.

6.2 Choosing a Population Size

In an ideal scenario, it would be best if the population size was in the hundreds. This would allow for more variation in the initial population and reduce the probability of the agents leading into local max instead of a global max. For example, if the initial population size is too low then the likelihood that an agent's inherited traits are the optimal solution is low as the genetic diversity is also low. However, during this project, the constraint that prevented us from running a population size that is in the hundreds was the computing resources available. When running a large population size, the game wasn't able to process every agent's location accurately which lead to agents dying when they should not have or living when they should have died. This makes sense because there is a lot to keep track of simultaneously including whether an agent collided with a pipe or the ground, the height at which an agent should flap, or the distance to the next pipe. With this in mind, the game seemed to run smoothly with a population size of 20 agents which is the size used in this experiment.

6.3 Choosing a Natural Selection Constant

When choosing a natural selection constantly, the choice determines the number of members from a population who gets to pass their genes onto the next generation of offspring. After running many different numbers we found that keeping 25% of the original population seemed like a valid solution. This meant that with a population size of 20 only 5 agent parameters would be used to create the next population of offspring. This percentage wasn't just chosen randomly. As the number of agents allowed to mate decreases, the rate of change between earlier generations increases. This means that the change between generations one and two will be more significant if only a select few individuals are able to mate. This means that only the most fit individuals can mate which leads to a larger shift in the population's parameter values. During this experiment, we tried to test a population by keeping 10 of the 20 agents. However, this led to the change between generations being too slow. This happened because it allowed for some agents that performed terribly to mate. This is why keeping the top 25% fittest agents seemed to work the best.

6.4 Running The Algorithm

Once the basic constants for designing a genetic algorithm were figured out the next part was to implement a working algorithm that would make each generation of agents better than the next. With the natural selection constant and the population size determined. The final step was to link the various steps of a genetic algorithm together to form a working solution. The steps that were taken followed this order; creating an initial population, running the population through the modified game, applying natural selection to the fittest agents, creating offspring, and repeating this process. In the code base, the steps to create a new population encompassed this new method.

```
getNewPopulation(scores, params, ns_const)
```

This method would take in the scores and input_params to the previous population along with the natural selection constant and apply both natural selection and mating to form a new population. This new population is returned and applied to the next generation of the game. This was the design behind implementing the genetic algorithm.

6.5 Setting Up Data Analysis

The last part of the project was to analyze the performance of the agents after they undergo the genetic algorithm. In order to design a system that could visualize the data, the performance and input parameters needed to be tracked through the different generations that were being run on the game. This was done through the use of a 3-dimensional array which stored the score, param1, and param2 for each of the 20 agents for each of the generations run by the program. Once, the program had finished the data could be visualized through a Python library called matplotlib. The various results from the program are described in the next section of the paper.

7 Results

Once the design of the project has been implemented, the last step of the project was to gather results. The results were based on a run of the game that evolved over the course of 10 generations. As described in the above section, the data collected had information about each agent. Figures 4, 5, and 6 show the different visual representations of the data collected. Also, in 3 you can see how the agent is able to perform well in the game from a genetic algorithm.

7.1 Trends in the Results

There are some important trends to notice about the data being displayed. As shown in Figure 4, the distribution of the two optimization parameters is randomly spread out and there is no clear distribution of the data. This makes sense as the original population parameters were chosen randomly. However, the graphs in Figure 5 show a different result. The data is more centered around a specific value.

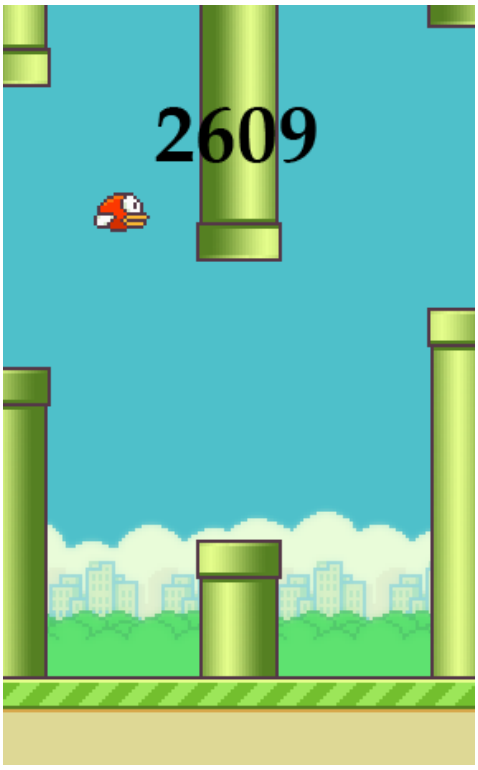

Figure 3. An agent performing well after the evolution due to evolution from the genetic algorithm.

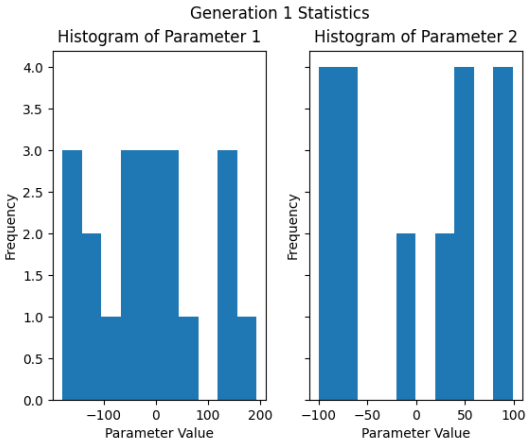

Figure 4. A histogram of the 2 parameters that are being optimized after the 1st generation.

7.2 Performance Across Multiple Generations

As the population evolves over across multiple generations, the score of the agents should increase. The observation from this trial regarding the score can be seen in Figure 6. This figure was created by taking the average score of all 20 agents at every generation in the trial and using it to create this graph. At first glance, it may seem like the evolution of

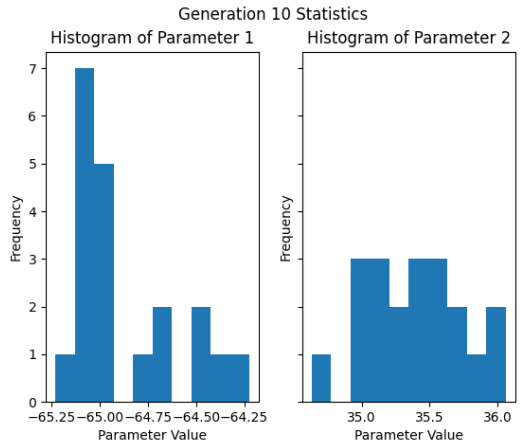

Figure 5. A histogram of the 2 parameters that are being optimized after the 10th generation.

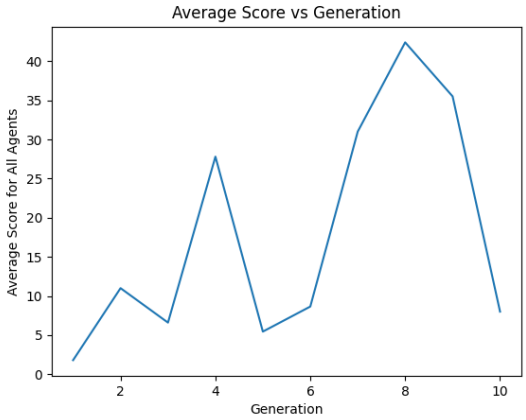

Figure 6. A graph showing the average score within a generation against the generations over time.

the agents is not increasing throughout every generation. Rather, it seems to improve in the beginning generations but then stays inconsistent in its performance. This phenomenon will be explained more in-depth during the analysis section of the paper.

8 Analysis

The results are interesting to look at, however, it takes some thought to analyze how they came to be.

8.1 Optimization Parameters

As shown in Figures 4 and 5 the data starts off being distributed randomly from roughly -200 to 200 pixels in parameter 1 and -100 to 100 pixels in parameter 2. Additionally, the performance with the values of these parameters is so spread out was bad. This leads to the graph of the plot of the

values from generation 10 in Figure 5. Notice that these values are much less spread out. The range between all the data points is less than 1 pixel in parameter 1 and slightly greater than 1 pixel in parameter 2. This means that an important thing occurred throughout the trial. The agent's parameters were successfully optimized through the use of a genetic algorithm. Based on the two graphs, the ranges between the agents' parameter values decreased significantly. This shows that the ideal value for the first parameter which looks at the position at which the bird should start accounting for the next pipe is about -65 pixels. This means that in the game the bird will start looking at the next pipe a little bit before it crosses through the closest pipe. The data seems counter-intuitive, yet it makes sense. Starting to analyze a pipe that is not even the closes pipe gives it more time to set up in the correct position after it crosses the current closest pipe. As far as the second parameter goes, the values that seemed to be the most effective were centered around 35-36 pixels. This means that the height at which the bird will constantly flap is if it is below the height of the pipe plus 35-36 pixels. This makes sense because the bird is in a good position for if the next pipe is lower or higher than the current position. By crossing the pipe toward the center it is able to move in a successful manner toward the next pipe. The process for which these parameters came to happen through the genetic algorithm as the better agents were able to score higher.

8.2 Performance

As shown in Figure 6. The performance of each generation doesn't necessarily increase. Yet, it does increase overall from the beginning generation but not from generation to generation. This seems odd until an analysis of the agents' gameplay. There are some errors in this version of the game's source code. This error involves the rate at which the new pipes spawn into the game. The pipes spawn slightly too fast and as a result, they are slightly too close to each other. This can lead to a rare case where if the pipes spawn in a specific combination, the agent is never able to advance to the new pipe. This will result in an automatic loss. This specific combination for a loss is if a pipe is low to the ground, then the second pipe is close to the ceiling, and a third low pipe is spawned. The agent is able to make it through the first two pipes but the final low-lying pipe is unbeatable. For this reason, an unlucky combination of pipes will guarantee all agents die no matter their parameter values. This issue doesn't affect the change in the parameters over time because they will always converge to the same values. However, it does hinder the ability to show a consistent increase in score over multiple generations. This was shown in Figure 6 and the average score was not consistently increasing. Based on that graph, it could lead one to think that the genetic algorithm's performance is not effective. Yet, this can be disproved by another graph displayed in Figure 7. In this graph, the initial starting population had a higher level of randomness ranging from -400 to 400 pixels for parameter 1 and -200 to 200 pixels for parameter 2. All of these ranges are double the values from the original -200 to -200 and -100 to 100 ranges. This leads to a more random starting population which will hence perform worse. The goal of this was to counteract the possibility of an unlucky pipe combination and force the population to change more dramatically between generations. Additionally, it would only show the early generations before the agents' parameters had started to converge. This scenario would help demonstrate the effectiveness of the genetic algorithm.

Based on the graph, the score of the agents' drastically increased throughout the first few generations. It seems like the parameters had already converged around the 5th and 6th generations. Nevertheless, it shows the effectiveness to find an agent's optimal parameters.

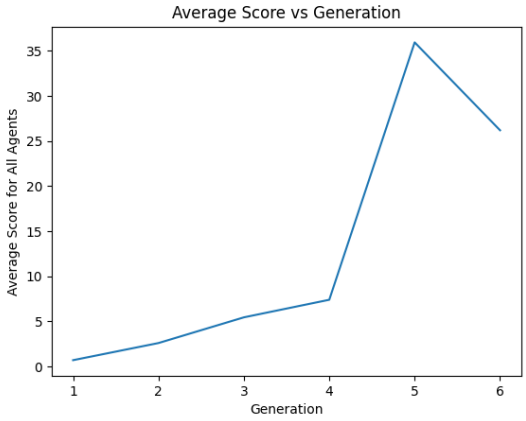

Figure 7. A graph showing the average score within a generation against the generations over time with a higher level of starting randomness.

9 Conclusion

The genetic algorithm developed for the problem space of Flappy Bird seems quite promising. Obtaining convergence in five generations of agents shows the immense learning speed of the algorithm and tenacity to solve problems given adequate fitness functions. In general, the success of fitness functions can be attributed to the choice of input parameters. The parameters can be tailored by understanding what success is best defined by and making that success quantifiable.

9.1 Improvements

The primary way to continue work on this agent would be to increase simulation accuracy to have a more accurate simulation for the game to remove the inaccessible parts of the map. This could be done by changing the way jumps

and gravitational acceleration such as having the nose of the bird tilt causing the way jumps interact with gravity. By making all environments more accurate to the original game agents could more accurately be evaluated in all situations. To improve the consistency of map difficulty, which can lead to misleading data as seen in Figure 7 would be to seed the pipe generation forcing pipes to generate in the same way for training causing map difficulty to be constant for training. The results could still be validated after removing the seed and testing on truly random environments. Another potential option would be to look at alternative algorithms discussed earlier such as the Neuroevolution of Augmenting Topologies (NEAT). By dynamic parameter optimization, this algorithm may not only reach optimum more quickly but do so with less computation overhead due to the simplicity of matrix operations needing to be performed for backpropagation. In testing the genetic algorithm developed could not run with a large number of agents and all potential input parameters, but by changing the evolutionary mechanism the speed could potentially be increased.

10 Contributions

- Abstract-Connor
- Problem Description- Both contributed to original reworked by Jonathan
- Background- Few changes to writing assignment 4 both contributed
- Approach - Connor wrote it ideas contributed by both
- Experimental Design - Connor
- Results - Connor
- Analysis - Connor
- Conclusion - Joanthan

References

[1] [n. d.]. https://www.codewithharry.com/videos/python-tutorials-for-absolute-beginners-122/

[2] Ahmed Fawzy Gad. 2021. How to train a game agent using the genetic algorithm. https://blog.paperspace.com/building-agent-for-cointex-using-genetic-algorithm

[3] John Henry Goldberg, David E.; Holland. 1988. Genetic Algorithms and Machine Learning. *Comput. Surveys* (1988).

[4] A. R. Kavitha and C. Chellamuthu. 2016. Brain Tumour Segmentation from MRI Image Using Genetic Algorithm with Fuzzy Initialisation and Seeded Modified Region Growing (GFSMRG) Method. *The Imaging Science Journal* (2016).

[5] Sourabh Katoch1 Sumit Singh Chauhan1 Vijay Kumar. 2020. A review on genetic algorithm: past, present, and future. *Multimedia Tools and Applications* (2020).

[6] S.J. Louis and C. Miles. 2005. Playing to learn: case-injected genetic algorithms for learning to play computer games. *IEEE Transactions on Evolutionary Computation* 9, 6 (2005), 669–681. https://doi.org/10.1109/TEVC.2005.856209

[7] Leonardo Vanneschi Mario Giacobini and William S. Bush. 2012. Evolutionary computation, machine learning and data mining in bioinformatics. Berlin ; New York : Springer, Article 10th European Conference.

[8] Aurimas Petrovas and Romualdas Bausys. 2022. Procedural Video Game Scene Generation by Genetic and Neutrosophic WASPAS Algorithms. *Applied Sciences* 12, 2 (2022). https://doi.org/10.3390/app12020772

[9] N A Phillip, S D H Permana, and M Cendana. 2021. Modification of Game Agent using Genetic Algorithm in Card Battle game. *IOP conference series. Materials Science and Engineering* 1098, 6 (2021), 62011.

[10] Lana Polansky. 2022. The leaderboard: The loneliness of the endless runner. https://www.pastemagazine.com/games/the-leaderboard-the-loneliness-of-the-endless-runn

[11] Douglas H. Werner Randy L. Haupt and Wiley InterScience. 2007. *Genetic Algorithms in Electromagnetics*. Hoboken, N.J.

[12] Kenneth O. Stanley. 2004. *Efficient Evolution of Neural Networks Through Complexification*. Ph. D. Dissertation. Department of Computer Sciences, The University of Texas at Austin. http://nn.cs.utexas.edu/?stanley:phd2004

[13] Junru Wang and Lan Huang. 2014. Evolving Gomoku solver by genetic algorithm. In *2014 IEEE Workshop on Advanced Research and Technology in Industry Applications (WARTIA)*. 1064–1067. https://doi.org/WARTIA.2014.6976460

[14] Rongbing Zhai. 2020. Solving the Optimization of Physical Distribution Routing Problem with Hybrid Genetic Algorithm. *Journal of Physics: Conference Series* 1550, 2 (may 2020), 022001. https://doi.org/10.1088/1742-6596/1550/2/022001