Comparative Analysis of Various Genetic Algorithm Approaches in Video Game Environments

Brendan Clare

Student, Connecticut College

Advisor: Stephen Douglass

12/10/2024

***Abstract*—This paper demonstrates the use of advances in the Genetic Algorithm (GA) to train autonomous agents to play video games, maximizing their scores. The algorithms compared are an implementation of the Neural Evolution of Augmenting Topologies (NEAT), a domain-independent classical genetic algorithm, and a coarse-grained parallel genetic algorithm. Data is collected for each algorithm based on their performance in three unique games. These games are my adaptation of popular browser/mobile games made in Processing.**

Keywords*—*Genetic Algorithms, Artificial Intelligence, Evolution, Learning, Fitness, Artificial Neural Networks, Neural Evolution of Augmenting Topologies, Parallelization

I. Introduction

GAs are efficient search-based methods inspired by biological evolution using principles of natural selection, crossover, and mutation. These algorithms can reach a good solution in a reasonable amount of time; however, more difficult problems may require unreasonable amounts of time or computing power. Thus, researchers are finding ways to make these algorithms more efficient and able to solve more complicated problems [4]. GAs have many real-world applications such as use in the fields of business, engineering, and science. They are also used frequently in video games to produce non-player characters, level

design, fix bugs, pathfinding, testing, speedrunning, and more.

The motivation behind this research is to compare how effective these algorithms are in a video game environment. Video games provide a controlled yet complex environment where the algorithms can be tested in real-time. By evaluating the performance of these algorithms, this research aims to better understand the strengths and limitations of each algorithm. Using a diverse environment of unique game-specific challenges will provide feedback on which situations each algorithm is best suited for. Ultimately this can be taken into consideration when using genetic algorithms for the various applications in video games.

II. Literature Review

Prior work demonstrates significant advancements in the computational efficiency of genetic algorithms. Katoch et al. (2021) discuss previous research on the different categories of genetic algorithms [5]. Cantú-Paz (2000) does a deeper dive into work on various parallel genetic algorithms [4]. Stanley and Miikkulainen (2000) present the method Neural-Evolution of Augmenting Toplopolgies (NEAT) which evolves artificial neural networks using the genetic algorithm [3]. Many studies also involve training video game AI using the genetic algorithm. Baldominos et al. (2015) describe competing in the Mario AI Championships, where programmers are tasked with training an autonomous agent to learn levels of the game Mario AI simulator based on Nintendo's *Super Mario World* to see which team can produce the best-performing agent on varying difficulty levels. They explored two approaches for parameters using the genetic algorithm: one in which knowledge of the game is considered and another domain-independent that could be applied to any problem [1]. Autin (2024) uses a library called *MarI/O* created by Seth Bling based on Stanley and Mikkulainen's NEAT architecture to train an agent to play *Super Mario Bros* gaining a general understanding of the game so that it can complete various levels without prior training on the specific level [2]. He aimed to improve the fitness calculation and create a dataset to train the agent.

III. Problem Description

What sets my research apart is training agents to play multiple games using three distinct approaches to the genetic algorithm. My goal isn't to refine the algorithms but to capture a more detailed picture of each algorithm's strengths, limitations, and overall effectiveness. This involves using existing libraries of

the genetic algorithms based on prior work discussed in this paper. This is to provide an even test environment so that performance results from previous advancements in each genetic algorithm.

IV. General Methods

The games I am using for testing are as follows: Flappy Tiles, 2048, and World's Hardest Game. To make these games I used Processing which is a Java-based free graphics library that provides an integrated development environment (IDE), a graphical user interface (GUI), and additional classes and mathematical operations. The algorithms used and implementations of the games will be discussed in this section.

A. Genetic Algorithm

GAs define finite-length strings which are possible solutions to a search problem referred to as *chromosomes,* the alphabets are referred to as *genes,* and the values of genes are referred to as *alleles.* The algorithm uses the following steps: 1. Initialization, 2. Evaluation 3. Selection 4. Recombination 5. Mutation 6. Replacement 7. Repeat 2-6 until a set stopping condition is met. Initialization involves defining the population size as well as the chromosomes. [6] The agents are evaluated based on the performance of the game they are playing using a fitness function. The fitness values are used to select the best players from the population. The best players are used as parents to produce offspring that recombine their solutions to produce better solutions potentially. There is a chance of mutation occurring by changing an individual's trait(s). Based on this selection, recombination, and mutation a new population replaces the previous generation. These steps are repeated until the program is manually terminated or a generation limit is reached.

B. Parallel Genetic Algorithm

Parallel genetic algorithms implement a divide-and-conquer approach, splitting up a task and solving it with multiple processors. There are different parallelization methods and one of them is the coarse-grained also known as multi-deme, distributed, or island parallel GAs. This version of the parallel genetic algorithm resembles the "island model" of Population Genetics where subpopulations of the same species are physically separated so they evolve separately. Each population of agents in the algorithm is treated separately, with the possibility of individuals migrating [5].

C. NEAT

NEAT is a Neuroevolution system introduced by Stanley and Miikkulainen in June 2002. NE approaches

evolve artificial neural networks with the genetic algorithm finding the networks that perform well at a given task. Neural networks are represented as genomes, where each genome is a set of genes. Genes can represent either a neuron (node) or a connection between neurons (edges). Traditional NE defines an initial fixed network structure where each neuron connects to each input and output. The goal of this is to improve the connection weights. NEAT differs in that the structure or the topology of the network can be changed which can also affect the performance. Through their research, they found that NEAT performs better than fixed topology networks in tasks such as the pole balancing problem. It starts with an initial population of simple neural networks. Networks are grouped into species based on similarity, and individuals within the same species can compete against each other. New nodes and connections can be added through mutation, and unnecessary structures can be removed to simplify the network.

D. 2048

Gabriele Cirulli originally developed 2048 in two days in March 2014 and has since had millions of downloads/plays. The game consists of a 4X4 grid of tiles. Using the arrow keys, tiles of the same value can be combined to merge into one tile, double the value of the original tiles. The player's score increases every time time tiles are merged adding the value of the new tile to the score. As the name suggests players try to get the 2048 tile, however, the game doesn't stop there and the player can keep playing making even greater tiles. The player loses when they can't possibly move in any direction as no adjacent tiles have the same value. The fitness function for this game is the player's score. My implementation of this game consists of a player class and a tile class. The tile class holds the value, x and y position, size, and color of the tile. Each player is initially given a 4X4 grid of these tiles and the value of each tile is 0 meaning it is an empty square. Also part of the setup is placing two tiles in random empty spaces with a ten percent chance of 4 and a 90 percent chance of 2. The autonomous agents use 4 outputs of the neural network to either move up, down, left, or right. Once the agent moves, a random tile is placed. The best player in the population is selected to be drawn at each frame, but the game is still keeping track of all the other player's grids and scores. The inputs for NEAT are the value of each tile, so there are 16 in total.

E. World's Hardest Game

World's Hardest Game was made by Snubby Land in 2008. It is a difficult 2D puzzle game where the player is a red square and they are trying to make it to the safe

zone while avoiding blue circles (enemies). The changes I made were the colors of the objects, the safe zone is a circle (goal) and the boundaries kill the player as this would be an inefficient route. My design for this game contains an enemy class that moves a circle up and down. The level class defines the boundaries, enemy starting positions, goals, and checks for collisions with the player. The player class sets the player's velocity and acceleration based on each algorithm. I used six vision inputs for NEAT: the player's x and y coordinate, the nearest enemy's x and y position, and the x and y coordinate of the goal. The fitness of the player is calculated based on the distance they are to the goal, but if the player reaches the goal the fitness is based on the amount of steps or frames it took for the player to get there. The game can be seen as a pathing problem where the players are trying to find the shortest route to the goal while avoiding the enemies. This is demonstrated below [8]:

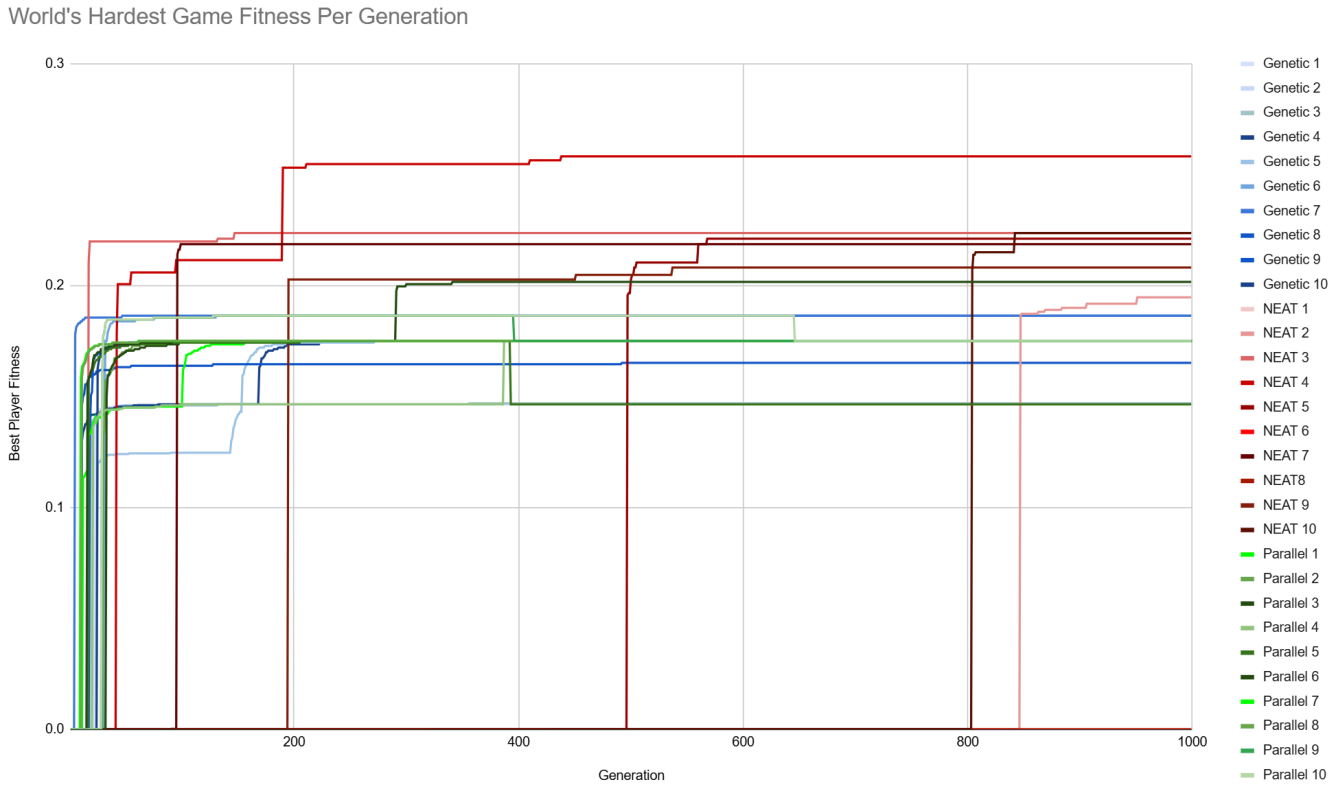
```
void calculateFitness(PVector goal) {
        if (reachedGoal) {
                fitness = 1.0/16.0 + 10000.0/(float)(step * step);
        } else {
                float distanceToGoal = dist(pos.x, pos.y, goal.x, goal.y);
                fitness = 1.0/(distanceToGoal * distanceToGoal);
        }
}
```

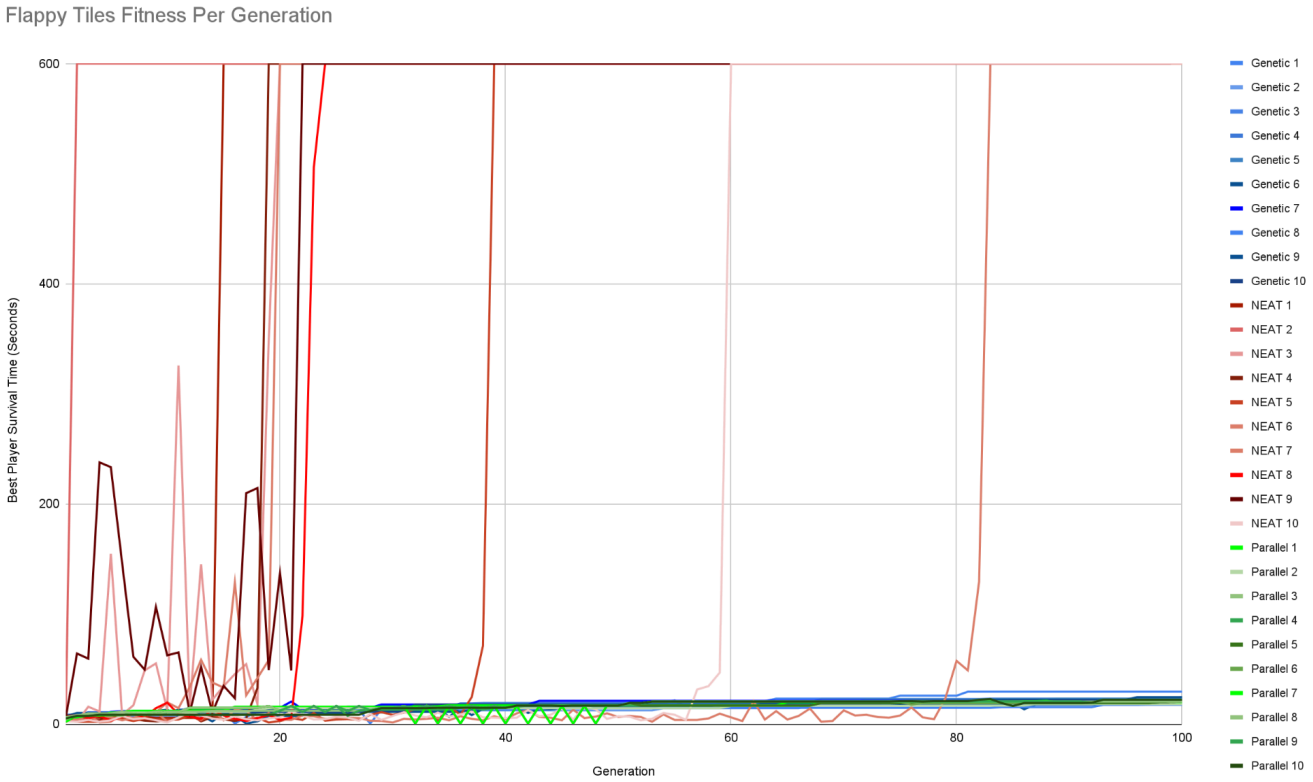
E. Flappy Tiles

"Flappy Tiles" is a game I created while taking the course *Introduction to Multimedia* at Queen Mary School of London. It consists of a player that is a green rectangle that moves up and down on the left side of the screen and enemies that are red rectangles that spawn at random y coordinates on the right side of the screen and move towards the left side of the screen (the player). The goal of the game is to survive as long as possible by avoiding the enemies. The fitness for this game is the score which is the survival time in frames. The vision inputs for NEAT are the player's y position and the nearest enemy's y position.

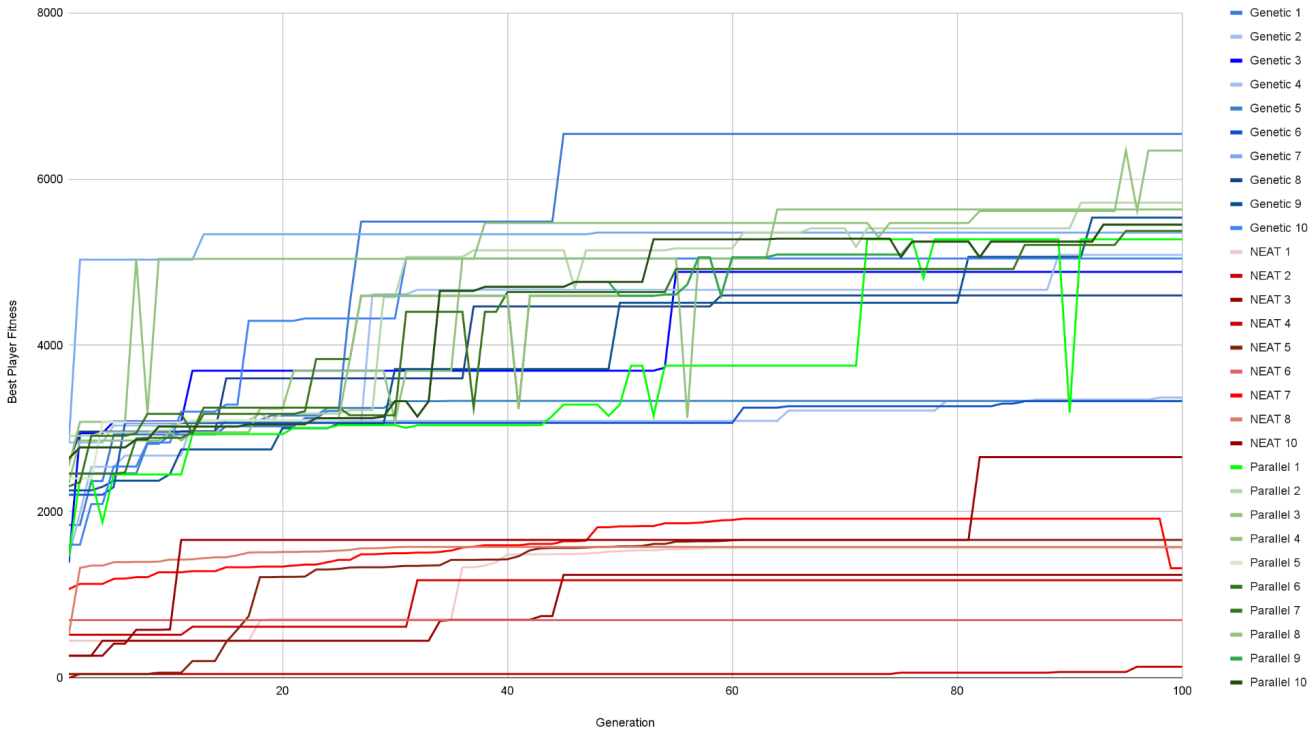
V. Results

For each game, 10 tests were performed due to the randomness involved in these algorithms. 1000 generations were tested for World's Hardest Game while 100 generations were tested for Flappy Tiles and 2048. Below are the graphs from these tests. Parallel is shown in green, genetic is shown in blue and NEAT is shown in red. Each test is a different shade of these colors:


World's Hardest Game Fitness Per Generation
Best Player Fitness
0.3
0.2
0.1
0.0
200
400
600
800
1000
Generation
Genetic 1
Genetic 2
Genetic 3
Genetic 4
Genetic 5
Genetic 6
Genetic 7
Genetic 8
Genetic 9
Genetic 10
NEAT 1
NEAT 2
NEAT 3
NEAT 4
NEAT 5
NEAT 6
NEAT 7
NEAT8
NEAT 9
NEAT 10
Parallel 1
Parallel 2
Parallel 3
Parallel 4
Parallel 5
Parallel 6
Parallel 7
Parallel 8
Parallel 9
Parallel 10


Flappy Tiles Fitness Per Generation
Best Player Survival Time (Seconds)
600
400
200
0
20
40
60
80
100
Generation
Genetic 1
Genetic 2
Genetic 3
Genetic 4
Genetic 5
Genetic 6
Genetic 7
Genetic 8
Genetic 9
Genetic 10
NEAT 1
NEAT 2
NEAT 3
NEAT 4
NEAT 5
NEAT 6
NEAT 7
NEAT 8
NEAT 9
NEAT 10
Parallel 1
Parallel 2
Parallel 3
Parallel 4
Parallel 5
Parallel 6
Parallel 7
Parallel 8
Parallel 9
Parallel 10

2048 Fitness Per Generation



VI. Conclusion

Based on the test data NEAT took longer to reach the goal in World's Hardest Game and in some cases, it never reached the goal after 1000 generations. However, once it reached the goal it was able to do it in fewer steps and find an overall more efficient path. Genetic and parallel on the other hand were able to reach the goal within about 20 generations however they stopped making as much improvement over time and NEAT was able to outperform them by the end. A trend across all the charts was that genetic and parallel perform largely the same and it is too close to determine which one is more optimal. In Flappy Tiles NEAT performed much better than genetic and parallel. It effectively mastered the game and had to be limited to 10 minutes or else it would keep going until eventually the game might crash. Genetic and parallel appear to make gradual improvements over time which is reflected in all of the graphs. In 2048 the opposite is the case–genetic and parallel outperform NEAT significantly. There are a few potential reasons for this. 16 inputs were used for 2048 NEAT which could be more than optimal. Finding the right balance between giving the neural network enough information and giving it too much where it takes too long to make the necessary connections is crucial for NEAT's effectiveness. Also, there might not have been enough generations tested due to the decision-based nature of this game. One important distinction is that genetic and parallel struggle immensely with randomness. They rely on a player's brain consisting of a string of decisions and if a certain player does well in one game, they are selected to reproduce. The issue is that with randomness a player that does well in one instance might do badly in another instance of the game. NEAT on the other hand is given input on these random events and is able to adapt its output or playing decisions accordingly. Thus, randomness was removed from all the games that genetic and parallel were tested on so they could produce better autonomous agents. All in all, the algorithm to choose for a given problem depends on what you want to prioritize and the specific problem. If you don't have as much time or processing power, it could be better to use genetic or parallel. If you want to find a more optimal solution, NEAT is a better choice. However, if you can't find adequate vision inputs for NEAT, genetic or parallel would be better. If there is any randomness involved, NEAT is significantly better.

VII. REFRENCES

[1] Baldominos, A., Saez, Y., Recio, G., & Calle, J. (2015). “Learning levels of Mario AI using genetic Algorithms.” In *Lecture notes in computer science* (pp. 267–277). https://doi.org/10.1007/978-3-319-24598-0_24

[2] Autin, Russell A., "Super Mario Evolution by the Augmentation of Topology" (2024). *University of New Orleans Theses and Dissertations*. 3161. https://scholarworks.uno.edu/td/3161

[3] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. Evolutionary Computation, 10(2):99–127, June 2002

[4] Cantú-Paz, Erick. “A Survey of Parallel Genetic Algorithms.” (2000). [PDF] A Survey of Parallel Genetic Algorithms | Semantic Scholar

[5] Katoch, S., Chauhan, S.S. & Kumar, V. A review on genetic algorithm: past, present, and future. *Multimed Tools Appl* **80**, 8091–8126 (2021). https://doi.org/10.1007/s11042-020-10139-6

[6] Sastry, K., Goldberg, D., Kendall, G. (2005). Genetic Algorithms. In: Burke, E.K., Kendall, G. (eds) Search Methodologies. Springer, Boston, MA. https://doi.org/10.1007/0-387-28356-0_4

[7] Code-Bullet. (n.d.). *GitHub - Code-Bullet/NEAT_Template: This is mainly for me, but if anyone wishes to use it then go ahead.* GitHub. https://github.com/Code-Bullet/NEAT_Template

[8] Code-Bullet.(n.d.).*GitHub-Code-Bullet/Smart-Dots-Genetic-Algorithm-Tutorial: Here is the code for my genetic algorithm tutorial*. GitHub. https://github.com/Code-Bullet/Smart-Dots-Genetic-Algorithm-Tutorial