Applying a Genetic Algorithm to Super Mario Bros.

Lucas Lenar
lenar067@umn.edu

May 14, 2019

Abstract

Genetic algorithms involve maintaining a population of agents which attempt to solve the given problem. Agents initially behave randomly, but over time the population may evolve to solve the problem effectively. In this paper, I will describe the modifications I made to an existing genetic algorithm for Arkanoid [9] to cater it to Super Mario Bros. I will also also analyze differences in how the genetic algorithm performs with parameters taking on different values and the possible reasons for its limitations. My genetic algorithm was written in Lua as plugin for the NES emulator FCEUX and maintains a population of individuals using a generational model.

1 Introduction

Evolutionary algorithms in general are interesting and important because they can be applied to many problems. Also, with enough time and data, they can exceed the performance of humans at certain tasks. In this project, I wanted to create a genetic algorithm to play Super Mario Bros. and see if it can approach the skill level of a human.

Each individual (or species, agent) in the population maintained by the genetic algorithm is defined by its chromosome, which influences in some way what inputs the individual executes while playing the game. As my implementation uses a generational model, the genetic algorithm allows each individual in the population to play Super Mario Bros. for some amount of time and then grades the individual on how it played, assigning it a fitness score. Once every individual in the population has played, the genetic algorithm allows the fittest individuals to survive into the next generation. These individuals then reproduce to replenish the population, after which all members of the population are subjected to random mutations. The process then repeats.

Making random changes to an agent may seem backwards or counter-intuitive to the reader unfamiliar with genetic algorithms. This process actually mirrors natural selection in real life, whereby individuals who are best adapted to their environment win the battle for resources and live long enough to reproduce, eventually causing populations to evolve. Random mutations are a real biological occurrence and are essential to a genetic algorithm, as they introduce significant variation into the population of agents. Some of these agents will perform better than others. This variation allows the genetic algorithm to select individuals for survival based on attributes that lead to better performance, as the agents with the best fitness are chosen to live on and reproduce. This populates the next generation with similar but different agents. Over many generations, the population of individuals evolve to improve at the given problem.

Applying a genetic algorithm to Super Mario Bros. is an interesting problem as it is a generally intuitive game for humans to play. I want to investigate how well a genetic algorithm can perform and compare the best agents it produces to that of a normal human player.

2 Related Work

Lee gives a good overview of genetic algorithms in [6], discussing their history and high-level ideas like reproduction and operators. A large section is devoted to the topic of encoding. A programmer has several options when it comes to encoding chromosomes, and Lee mentions that encoding mainly depends on the problem. Binary encoding is very common because it was the first encoding used for genetic algorithms. However, binary encoding is often not natural for many problems [6]. Lee also briefly covers different representations of chromosomes such as trees, arrays, and lists as well as high-level implementations of mutation and crossover operations on these representations.

In [7], Perez et al. focus on their application of Grammatical Evolution to evolve Behavior Trees to create controllers for the Mario AI Benchmark, a software that was used in the 2010 Mario AI Competition. The authors used a genetic algorithm to evolve variable-length integer strings, which are then mapped to phenotype programs based on a context-free grammar [7]. These phenotype programs are Behavior Trees, which consist of different types of nodes, the most basic of which either check a condition such as the presence of enemies or execute an action. Perez et al. had success in their experiments, and found that their approach led to very good reactive behavior and strengthens the idea that Genetic Programming systems are serious alternatives to more traditional AI algorithms [7].

In [4], Hou et al. discuss their implementation of an AI for Infinite Mario Bros. (a variation of Super Mario Bros.) using a combination of a Genetic Algorithm and a Finite State Machine (FSM) with the states: Run, Run Jump, Run Speed, Run Jump Speed. The FSM changes its current state based on the individual's chromosome and whether it sees an enemy, obstacle, or pit [4]. The authors found the AI controllers that evolved were capable of playing Infinite Mario Bros. well and could find optimal solutions. Hou et al. also discuss some useful metrics they used in their experiments like crossover rates, mutation rate, individuals per generation and total generations.

In [3], Flom & Robinson discuss applying a genetic algorithm to Tetris. In their implementation, each agent has its own evaluation function that determines how it plays. The evaluation function's weights are applied to aspects of the current game state such as pile-height, closed holes, wells, number of lines that were just made, and how "bumpy" the pile is. These weights are determined by the agent's chromosomes. The authors tested agents using the same sequence of pieces so comparison would always be fair, and calculated each agent's fitness based on how many lines the agent made in a certain time period as well as the ratio of lines made to pieces played [3]. Flom & Robinson had promising results. The best agents produced by their implementation performed very well after 1000 generations.

Ponsen et al. employed a more complicated evolutionary algorithm in the real-time strategy (RTS) game Wargus using an evolutionary state-based tactics generator (ESTG) and dynamic scripting in [8]. The authors describe dynamic scripting as a reinforcement learning technique designed for creating adaptive video game agents and focus on dynamic scripting because it can generate a variety of behaviors and respond quickly to changing game dynamics [8]. The authors specify a small number of states a game of Wargus can be in. The agent that is controlled by dynamic scripting then chooses its tactics based on the knowledge base for the current state of the game. Dynamic scripting maintains a weight value for each tactic in a state-specific knowledge base. Each tactic's weight indicates the desirability of choosing that tactic in the specific state, and the weight values of all employed tactics are updated based on how the agent performed [8]. The knowledge bases used by dynamic scripting are created via the process of ESTG, in which an evolutionary algorithm creates counter strategies to a training set of strategies. Knowledge transfer then transfers the evolved strategies to tactics based on states to create the knowledge bases. The authors found their use of dynamic scripting defeated top strategies submitted for a Lehigh University tournament and generalized against strong strategies not in the training set [8].

In [2], Chalmers focuses on a very different problem from those previously mentioned: the evolution of a learning procedure so an agent can perform multiple tasks rather than one, as in [8], [3], [7], and [4]. Chalmers does this through genetic connectionism on the adaptation of neural networks to an environment (a learning task) by interacting with an evolving learning procedure. His goal is to develop learning procedures that can be applied to initially random neural networks to teach them a learning task [2]. It is important to qualify what Chalmers means by random. At several points in his paper, Chalmers makes the distinction that the genome of a learning procedure encodes how to change the connection strengths of a neural network, meaning only the dynamic properties of a neural network are affected or even known in the learning procedure. This is why Chalmers clarifies that the neural networks are initially random within a constrained framework. Chalmers found that learning algorithms evolved general learning mechanisms when the evolution environment was sufficiently diverse: when there were 10 or more learning tasks, neural networks that interacted with the best learning algorithm (determined after 1000 generations) performed very well on learning tasks not in the evolution environment [2].

In [5], Janikow and Michaelewicz compare binary and floating point representations in genetic algorithms. The authors argue that a floating point implementation may result in much better performance and precision. They compared a floating point implementation with new operators to a binary implementation on a test case [5]. The authors first compared mutation in the binary and floating point implementations and found the floating point representation has a drawback of mutation being "more" random. After introducing a new dynamic mutation operator, they found the floating point implementation had a better average performance than the binary implementation as well as having more stable results. Janikow and Michaelewicz found in their experiments that the floating point representation is faster, more consistent across runs, and has higher precision, especially in problems with large domains where a binary representation would be very long [5].

Vavak and Fogarty also compare different approaches to genetic algorithms in [10]. They compare two models of genetic algorithms: generational and incremental/steady state and used a non-representative problem (bit-matching) to compare the two models. The generational genetic algorithm creates new offspring from members of the old population using genetic operators, whereas the incremental/steady state model involves inserting one new member into the population at a time in place of one that is deleted. Vavak and Fogarty found the incremental model outperformed the generational model, finding the optimum in a smaller number of evaluations. They argue the incremental model has better performance because offspring are immediately used as part of the mating pool, making shifts toward the optimal solution possible relatively early [10].

3 Approach

While researching genetic algorithms for video games, I found an existing genetic algorithm [9] for the NES game Arkanoid (similar to Atari's Breakout). This provided a very good framework for my genetic algorithm. [9] uses a binary encoding for chromosomes, where an individual's chromosome simply represents the moves the agent will execute with its paddle. A '0' indicates the agent will move its paddle to the right and vice versa. Individuals start with a chromosome size of 15, and inputs are made for 20 frames each, so agents can initially play for 300 frames.

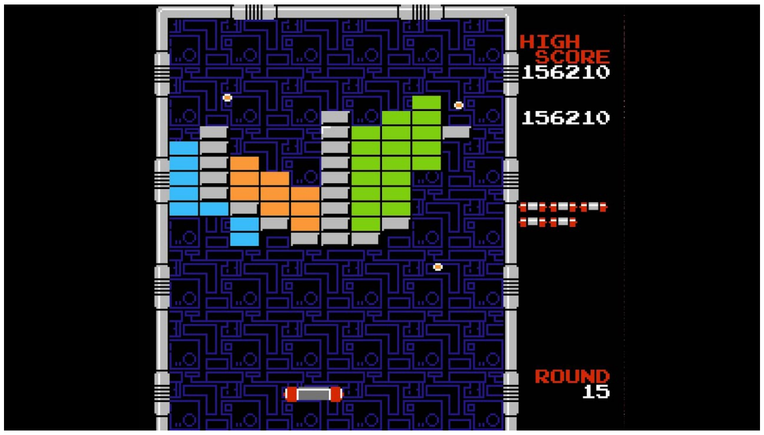


Figure 1: The NES game Arkanoid

Three parameters that affect the genetic algorithm are: **control_gap**, **crossover_rate**, and **mutation_rate**. After each generation, the crossover operation adds **control_gap** random bits to the top **crossover_rate**% performers. The default value for **control_gap** is 5. It is intended to be small so that good adaptations are made at each step before chromosomes grow too large. Most of the remaining members of the population are offspring of the top **crossover_rate**% of performers, each created by appending the starting **control_gap** control bits of some top performer to another (determined randomly). The last 10 members are completely random to help keep diversity in early generations. Mutation iterates through the members of the population and has a **mutation_rate**% chance of flipping each bit.

I decided to take this framework and modify it to fit Super Mario Bros. I initially wanted to implement an action function that each individual would use to react to the current game state. This would have involved using a hash function to map the chromosome of the current species and the current RAM of the game onto an element of the power set of possible inputs. I was unsuccessful with this, and this would be more akin to a neural network. I therefore decided to instead use the idea in [9], where a chromosome represents a sequence of inputs. Because Super Mario Bros. allows significantly more meaningful simultaneous inputs than Arkanoid, this required modifying the encoding of a chromosome.

Instead of each bit in an individual's chromosome encoding a move to the left or right, my genetic algorithm uses a concept similar to genes, where genes can be thought of as a sequence of eight bits in a chromosome. Each bit in a gene corresponds to one of the buttons on an NES controller. A '1' means the corresponding button will be held down. The mutation function for chromosomes of this type is logically equivalent to that of [9]. Each bit in each gene of an individual's chromosome has a chance to be flipped from '1' to '0' or vice versa.

Although the start and select buttons are not needed for normal play in Super Mario Bros., and one could argue these should have been left out of the encoding for efficiency (as pressing start pauses/unpauses the game), I decided against this as I viewed this as an unnecessary simplification. I also view agents evolving to pause less frequently as nontrivial and interesting. Preventing agents from pressing left/right or up/down simultaneously was, for the same reasons, not done.

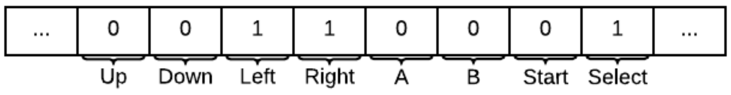
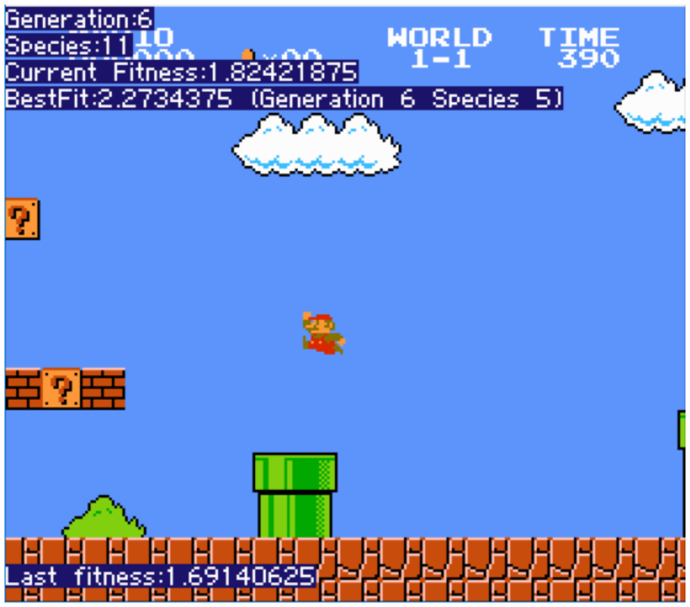


Figure 2: Example gene in a chromosome of some length

A surprising modification that needed to be made to [9] to allow evolution was the removal of the logic in [9] that ensures buttons are held down for 20 frames. It is intended to ensure smooth movement. However, this stunted evolution because agents were unable to execute jumps of variable length (requiring holding A for a variable number of frames). Therefore, each gene corresponds to the inputs that will be executed for exactly one frame. Because my implementation of genes requires eight bits rather than one and a gene represents inputs to be executed for only one frame rather than 20, the chromosome size and **control_gap** needed to be increased significantly.



```
Generation:6
Species:11
Current Fitness:1.82421875
BestFit:2.2734375 (Generation 6 Species 5)
Last fitness:1.69140625
```

Figure 3: My genetic algorithm in progress

The fitness formula also needed to be modified. I used a calculation including the current world, level, and x position when the agent died. This required reading the RAM of the game. For this purpose, the Super Mario Bros. RAM map on datacrystal.romhacking.net [1] was an extremely useful resource. This fitness function is intended to be monotonically increasing for normal play so agents do not converge to local maxima.

I also tested a modified version of the original mutation function that does not mutate the top performers in a generation. My idea was that the best performers of a generation should be preserved, and we should not risk losing the progress made by those top performers. This would be especially important at bottlenecks in later generations. I will compare this change along with others in the next section.

4 Experiment Design and Results

For each of my experiments, I ran the genetic algorithm for at least 20 generations using various values for **control_gap**, **mutation_rate**, and **crossover_rate**. I also ran the genetic algorithm with the modified mutation function mentioned previously. All experiments were carried out with **chromosome_size** = 1000 and **control_gap** = 1000.

This experiment is intended to better understand how the genetic algorithm works and what values might be better for these parameters to increase performance. It will also inform whether my theory on preserving the top performers without mutation is valid. What follows are graphs displaying the effects of the modified mutation function and altering **mutation_rate** and **crossover_rate**. Also included are graphs displaying the average and best fitness of all experiments.

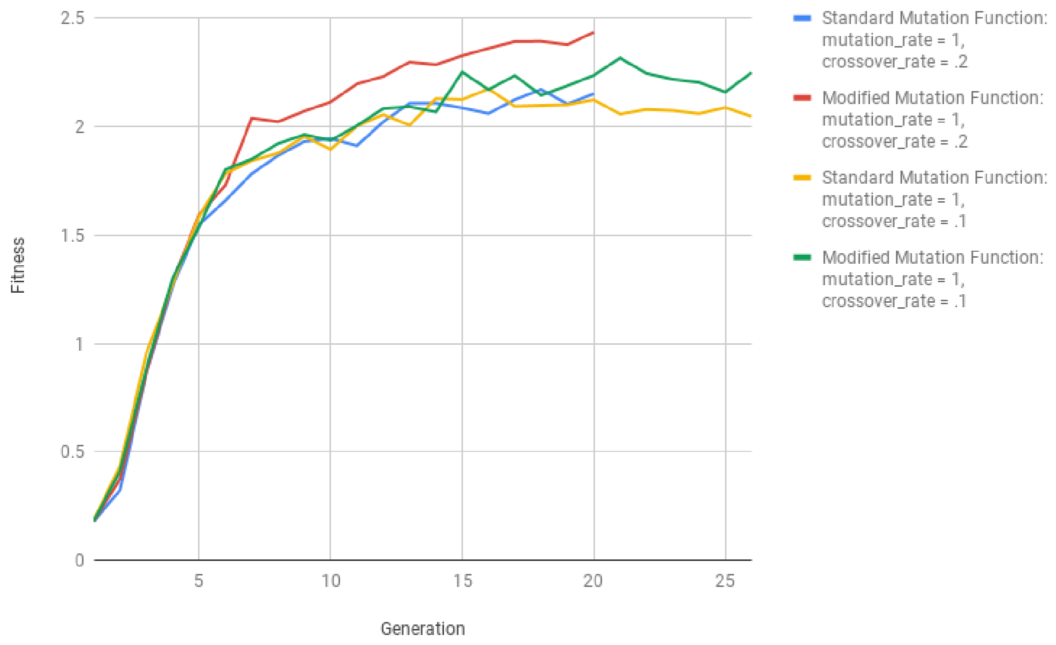


Figure 4: Effect of Modified Mutation Function on Average Fitness

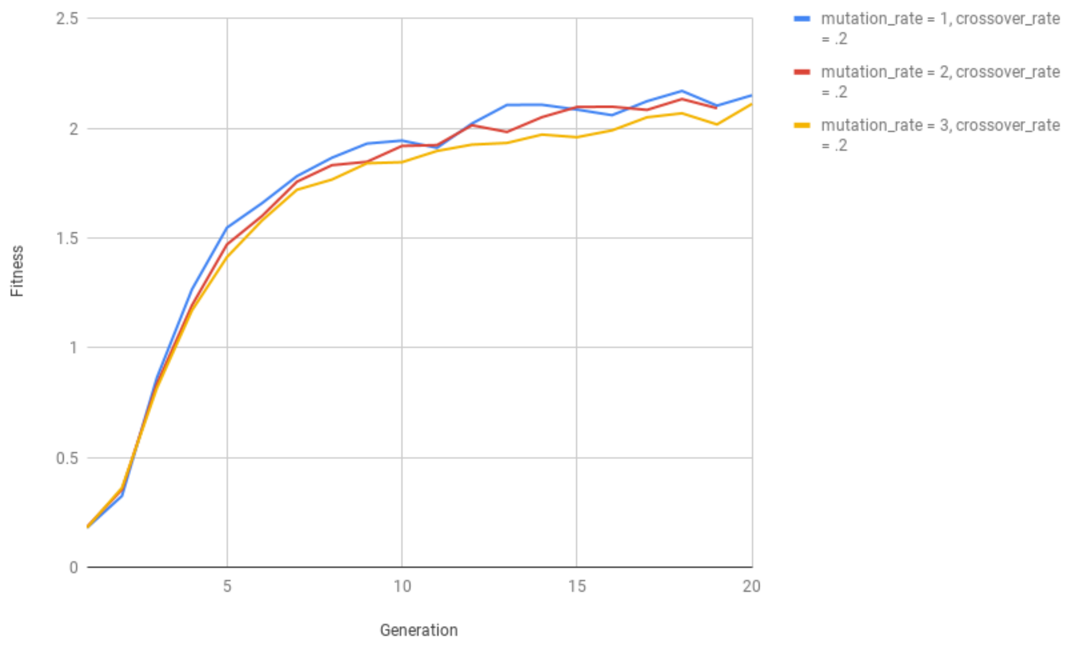


Figure 5: Effect of Mutation Rate on Average Fitness

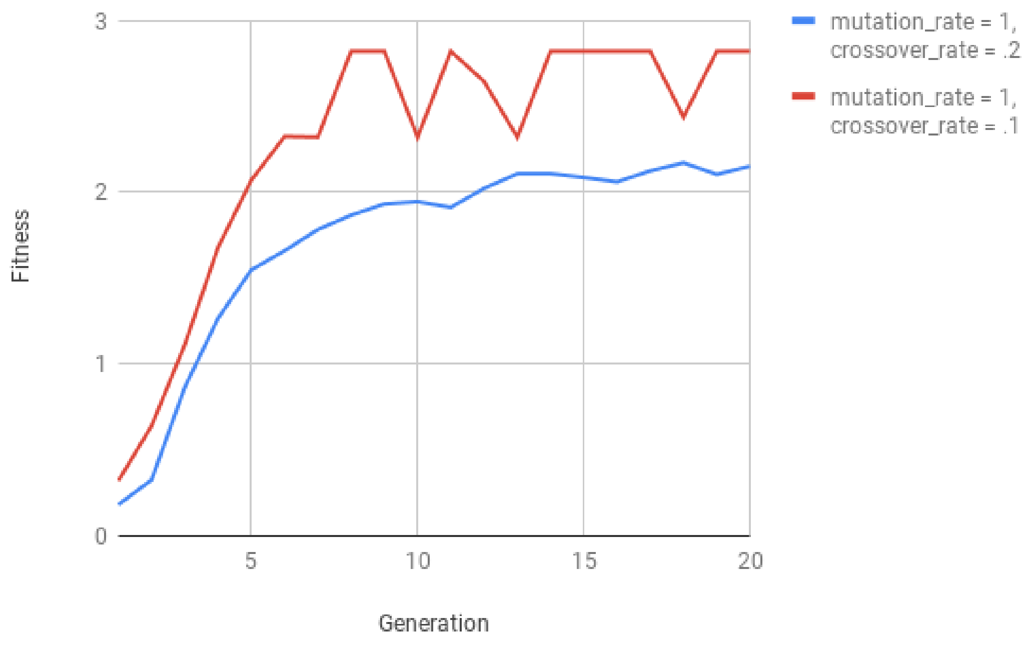


Figure 6: Effect of Crossover Rate on Average Fitness

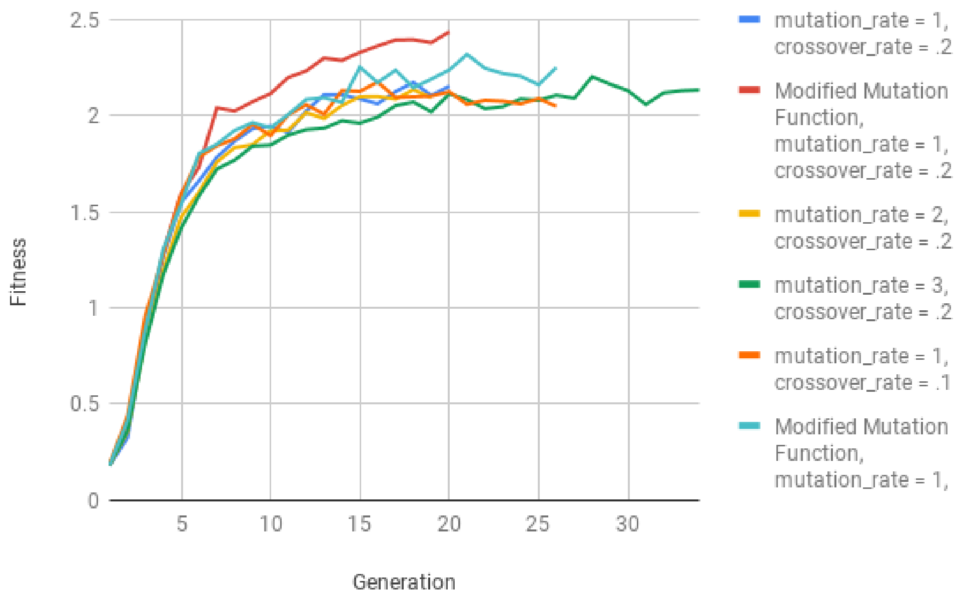


Figure 7: Average Fitness of All Experiments

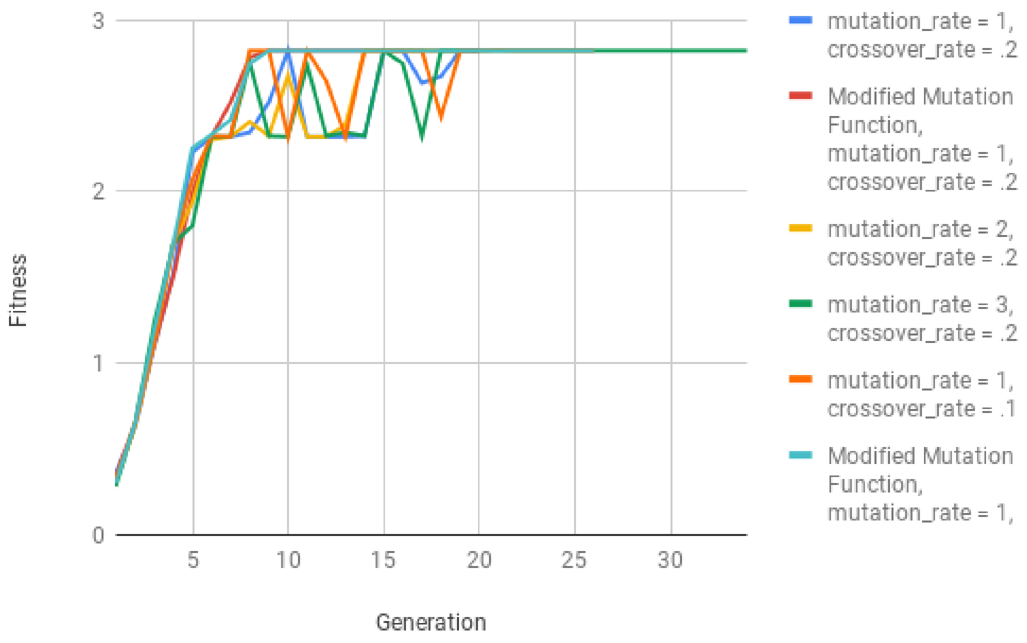


Figure 8: Best Fitness of All Experiments

5 Analysis of Results

As can be seen in Figure 4, my modified mutation function was successful. In both instances where it was tested, it improved average fitness in later generations without diminishing improvement in earlier generations due to decreased variation in the population. Also, as can be seen in Figure 7, the experiment with the best average fitness by far was one using the modified mutation function.

From Figure 5, we can gather that a mutation rate of 3% or higher negatively affects the performance of the genetic algorithm. With **mutation_rate** = 3, a gene in a chromosome will be altered with probability $(1 - .03)^8 = .216$. At this point, mutation leads to excessive randomness in chromosomes, causing individuals to lose beneficial genes and slowing evolution. We may even be able to make a stronger claim that a mutation rate of 2% or higher negatively affects performance as the experiment with **mutation_rate** = 1 has the best average performance in most generations once the experiments diverge.

Although Figure 6 is only a comparison between two experiments, it does show that crossover rates of 20% or higher negatively affect the performance of the genetic algorithm. It makes sense that being selective regarding how many individuals are allowed to survive into the next generation would benefit fitness over many generations. The average fitness can be seen to be more erratic across generations when **crossover_rate** = .1, whereas average fitness increases much more smoothly when **crossover_rate** = .2. This may be because allowing only the best 10% of individuals to survive eliminates otherwise good adaptations that existed in the next 10% of individuals, which are allowed to survive when **crossover_rate** = .2. Although having a crossover rate of 10% leads to higher performance, it is also more inconsistent. I will discuss the optimal crossover rate more in the Future Work section.

I will now discuss general trends seen throughout the experiments, for which Figures 7 and 8 are useful. They show that most experiments had very similar and rapid improvements in both average and best fitness for the first 5 generations. In all experiments, individuals adapted to move to the right after just one generation. A curious behavior observed even in later generations is all individuals jumping erratically. This is not very surprising, as there is no significant selective pressure to prevent this behavior.

Despite rapid improvements in early generations, the best fitness eventually plateaued severely in all experiments, sometimes remaining static for over 10 generations. This point in the game was the third pipe in the first level, which no agent ever passed. This puzzled me, so I decided to run an experiment to estimate how long an agent would need to hold down the A button to jump over the pipe, which came to be about 20 frames.

This would mean that for an individual to pass this pipe, its chromosome would need to consist of about 20 consecutive genes with the bit corresponding to the A button being '1', followed by a gene with the bit corresponding to "Right" being '1'. However, genes with the A button bit being '1' will not contribute to the agent jumping over the pipe if the game is paused, as Mario's progress is halted. Therefore, passing the pipe would require either a great deal of order between A being held down and start not being pressed or an even greater number of consecutive genes with the A button bit being '1', both of which one would only expect to come about after many generations.

I think it is important here to discuss the generational scale of my experiments compared to others in the relevant literature. In [3], Flom & Robinson's experiments ran for between 700 and 1000 generations. Also, significant relative improvement in the first 50 or so generations of their experiments cannot be seen. Although they used a different implementation with an evaluation function, I believe this is still relevant information. As I did not have the resources to run many experiments with over 700 generations, I am not discouraged by the somewhat underwhelming performance of my genetic algorithm after 20-40 generations. I am still confident in the robustness of my implementation and that some agent would eventually pass this bottleneck point.

6 Conclusion and Future Work

Although the performance results of my genetic algorithm were somewhat disappointing, I am still confident in its validity. The modifications I made to [9] were interesting, including the introduction of gene encoding for eight simultaneous inputs and the modified mutation function that allows the best individuals to survive without being mutated. My project as a whole qualifies as a proof of concept with potential to have much better performance in the future.

In the future, I would like to run more experiments to find the optimal value of **crossover_rate**, as I find the idea of there being an optimal percentage of individuals who should be allowed to live on and reproduce very interesting. I would also be interested in allowing my genetic algorithm to run for many generations (on the scale of [3]) with the best settings.

A significant and useful modification to my project would be using the incremental/steady state model mentioned in [10], considering the definite performance benefits brought up by Vavak and Fogarty. The use of Grammatical Evolution and Behavior Trees in [7], as well as the type of learning procedure discussed by Chalmers in [2] would also be very interesting. All three warrant pursuit in future work I do on genetic algorithms either in this problem or in other domains.

[6] is a useful resource for the reader previously unfamiliar with genetic algorithms who is interested in a beginner's overview. [5] and [10] may prove interesting to the reader as discussions of models and representations of genetic algorithms they are unfamiliar with. If the reader found this paper enjoyable, I recommend all other related works as well.

References

[1] Super mario bros.:ram map.

[2] D. J. Chalmers. The evolution of learning: An experiment in genetic connectionism. In *Connectionist Models*, pages 81–90. Elsevier, 1991.

[3] L. Flom and C. Robinson. Using a genetic algorithm to weight an evaluation function for tetris, 2005.

[4] N. C. Hou, N. S. Hong, C. K. On, and J. Teo. Infinite mario bross ai using genetic algorithm.

[5] C. Z. Janikow and Z. Michalewicz. An experimental comparison of binary and floating point representations in genetic algorithms. In *ICGA*, pages 31–36, 1991.

[6] S. K. Lee. Genetic algorithms.

[7] D. Perez, M. Nicolau, M. ONeill, and A. Brabazon. Evolving behaviour trees for the mario ai competition using grammatical evolution.

[8] M. Ponsen, H. Munoz-Avila, P. Spronck, and D. W. Aha. Automatically generating game tactics through evolutionary learning. *AI Magazine*, 27(3):75–75, 2006.

[9] SurenderHarsha. Surenderharsha/arakanoid_genetic_algorithm, Jun 2018.

[10] F. Vavak and T. C. Fogarty. Comparison of steady state and generational genetic algorithms for use in nonstationary environments. In *Proceedings of IEEE international conference on evolutionary computation*, pages 192–195. IEEE, 1996.