NORDIC MACHINE INTELLIGENCE

Nordic Machine Intelligence, vol. 10, pp. 1–19, 20XX
Received 1 Jan 20XX / published 21 Jul 20XX
https://doi.org/XX.XXXXX/nmi-some-doi

Optimizing Mario Adventures in a Constrained Environment

Sanyam Jain[1]

1. *Østfold University College, Halden, Norway, sanyamj@hiof.no*

Abstract
This project proposes and compares a new way to optimise Super Mario Bros. (SMB) environment where the control is in hand of two approaches, namely, Genetic Algorithm (MarioGA) and Neuro-Evolution (MarioNE). Not only we learn playing SMB using these techniques, but also optimise it with constrains of collection of coins and finishing levels. Firstly, we formalise the SMB agent to maximize the total value of collected coins (reward) and maximising the total distance traveled (reward) in order to finish the level faster (time penalty) for both the algorithms. Secondly, we study MarioGA and its evaluation function (fitness criteria) including its representation methods, crossover used, mutation operator formalism, selection method used, MarioGA loop, and few other parameters. Thirdly, MarioNE is applied on SMB where a population of ANNs with random weights is generated, and these networks control Mario's actions in the game. Fourth, SMB is further constrained to complete the task within the specified time, rebirths (deaths) within the limit, and performs actions or moves within the maximum allowed moves, while seeking to maximize the total coin value collected. This ensures an efficient way of finishing SMB levels. Finally, we provide a five-fold comparative analysis by plotting fitness plots, ability to finish different levels of world 1, and domain adaptation (transfer learning) of the trained models. Overall, MarioGA was able to finish levels 1-1,1-2,1-3 and 1-4 XX% times where as MarioNE was able to finish XX% times with fitness of XX and XX respectively. The average time taken by MarioGA to finish each level is XXXX seconds with XXXX distance, and XXXX coins collected. However MarioNE takes XXXX seconds to cover XXXX distance with XXXX coins to finish the same levels.

Keywords: Genetic Algorithm, Neuro-Evolution, Super Mario

Introduction

In agent-based systems (for example, Robotics) it is critical to have an efficient agent which can preform dedicated tasks in constrained environment optimally. To formalise such tasks in a simulated environment, we choose "gym-super-mario-bros" library [1] which is an open-source Python package that extends the Gym framework to provide a framework for experimenting with SMB. In order to achieve a task-based control in such constrained environment or dealing with this constrained optimisation problem (COP) we propose MarioGA and MarioNE which implements and evaluates the SMB environment such that our agent (Mario) could complete levels within time limits and collecting as many coins along with distance covered. We further discuss lot more details why we hypothesise such fitness function in following sections.

A detailed SMB environment variables are presented in Table 1 SMB environment has been studied a lot in different setups recently [2, 3]. For example in [4] has provided a experimental study of Proximal Policy Optimization (PPO), Deep Q-Network (DQN), and their proposed approach of Reptile algorithm (a.k.a RAMario). However, our focus restricts more towards Genetic Algorithm (MarioGA) and Neuro-Evolution (MarioNE). We present and study more about previous works in GA and NE in next section.

Based on the availability of input, output and model, there is a classification for Optimisation, Modelling and Simulation in [5], our problem fits with Optimisation. Additionally, we also know this is a COP because we have output specified as the best path that Mario traverses such that it is travels more and collects maximum coins in its adventure, keeping with constraints of time, actions or movement done, etc. So the output is specified however we do not know exactly what input combinations will give this. Further, for a given instance of environment, we can always know (using a formula or calculations) how many coins it has collected so far at the moment, distance traveled, movements or actions performed, number of rebirths, and time elapsed so far at the moment. Hence model is also defined. We formulate a COP over state space S, defining $\psi(s)$ as the constraint function, ensuring satisfaction of specified constraints, and introducing a new function $g(s)$ that quantifies constraint violations; thus, a game configuration represents a solution to the Mario COP only when $g(s) = 0$ and $\phi(s) = true$, signifying successful coin collection, distance maximization in minimum game time, and constraint adherence. In addition, the problem falls under constrained optimization and is classified within NP due to its efficient verification process for valid solutions, making it suitable for NP algorithms like Evolutionary Computation, particularly when verifying if Mario's path adheres to constraints while collecting at least X coins.

Table 1: Mario Game Dictionary

Key	Type	Description
coins	int	The number of collected coins
flag_get	bool	True if Mario reached a flag or ax
life	int	The number of lives left, i.e., {3, 2, 1}
score	int	The cumulative in-game score
stage	int	The current stage, i.e., {1, ..., 4}
status	str	Mario's status, i.e., {'small', 'tall', 'fireball'}
time	int	The time left on the clock
world	int	The current world, i.e., {1, ..., 8}
x_pos	int	Mario's x position in the stage (from the left)
y_pos	int	Mario's y position in the stage (from the bottom)

MarioGA uses standard genetic algorithm described in the textbook "Introduction to Evolutionary Computing" [5]. A standard MarioGA approach is inspired from nature or natural evolution such that we have an environment (SMB) which is filled with candidate solutions (population) that strive for survival and their reproduction (crossover) where their quality (fitness) is determined by environment which provides a criteria whether they (elites) or their children (crossover and mutation) are going to grow in the environment which are fighting for resources (as metaphor) most effectively. Two key terminologies, Genotype and Phenotype (also discussed later in further sections) are core to any individual in the process, where for us in SMB world, actions, movements (when to jump, move left, move right, and use power-ups), and other Mario's game-play strategies can be classified as genotype where as the resultant behaviour of those encoded action plan is phenotype. To name a few phenotype, it can be, Mario navigating the level, jumping over obstacles, collecting coins, defeating enemies, and completing the level. Here it is important to note that, one possible value of genotype is allele (for example, In this case, the gene associated with Mario's jumping behavior has two possible alleles: "High Jump" and "Low Jump."). An important point to note in this discussion is that phenotypic differences always stem from variances in the genotype, which can further be simplified that genotype encodes the material which abstracts the information necessary for a particular phenotype. We envision to also use crossover which is an operation mimics genetic recombination inspired by meiosis, where genetic information from two parent solutions is combined to produce offspring solutions with diverse traits. The reason or motivation why we propose MarioGA (Evolutionary Algorithm or Genetic Algorithm) is because of the complexity of the problem to be solved. Since in this COP, we do not have much options to tailor or handcraft, which makes it impossible to try out all possible combinations of the game configurations to be used. Since, EAs or GAs are proven to be robust in such cases to provide near-optimal or optimal solutions as given examples in [5].

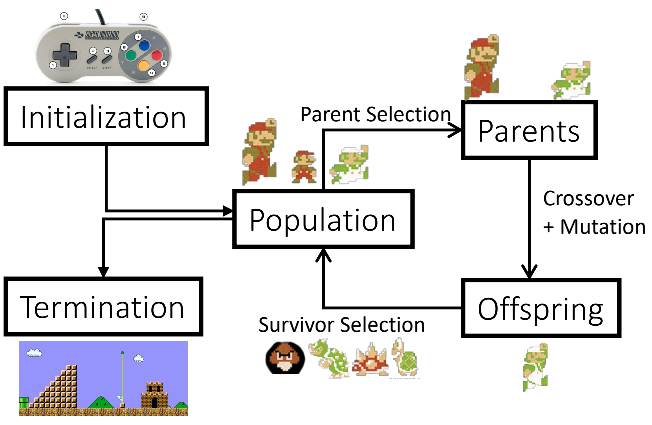


Figure 1: Standard Genetic Algorithm

MarioNE uses standard Neuro-Evolution approach described in [6]. MarioNE utilizes artificial neural networks (ANNs) to enable Mario to learn and adapt his gameplay strategies. Think of it as teaching Mario to become a better player by allowing him to practice and evolve within the game. This approach draws inspiration from nature, where animals strive to survive and reproduce, gradually improving their abilities. MarioNE aims to optimize how Mario plays the game, enhancing his skills in actions like jumping, coin collection, and level completion. It is much simpler and relaxed algorithm comparing to EA or GA such that NE only requires a measure of a network's performance in a given task. For instance, it can evaluate the outcome of a game, such as whether Mario won or lost, without the need for labeled examples of desired strategies [7].

In following text and further sections, we study Related Works, Proposed Methods (MarioGA and MarioNE), Implementation Details, Results, Discussion and finally Conclusion. Additionally, we study Initialisation, Representation of individuals, Evaluation or Objective function, Population, Selection criteria, Variation operations (Crossover and Mutation) and finally Termination criteria as part of explanation for MarioGA as shown in Figure 1. A very similar pipeline is followed by MarioNE however, rather than initialising agent that can perform various Mario operations (configuration), we initialise ANNs that controls SMB and fitness is evaluated on the network's performance on evaluation criteria.

Related Work

We study two work that are focused in utilising GA and NE to play SMB respectively. Among the approaches in [2, 3, 4, 7], GA leverages evolutionary principles to optimize SMB's gameplay sequences, evolving action plans over generations. On the other hand, NE employs artificial neural networks to evolve intelligent controllers for Mario, allowing him to adapt and learn in dynamic environments.

GAs optimize SMB's gameplay by evolving action sequences over generations, representing potential paths that undergo mutation and crossover for enhanced performance. Fitness evaluation considers coin collection, distance traveled, and constraints, ultimately guiding Mario to collect more coins efficiently in constrained game environments. GAs simulate natural selection, starting with random sequences and selecting the best for reproduction. Over time, these sequences improve, optimizing SMB's gameplay strategy [8]. They propose a very good setup, however they consider fitness as $Distance + (20 * Coins)$ which makes it very ineffective for an agent to travel longer distances. For example, consider for example, a situation in the game where there is a place with a pipe and collection of coins (as shown in Figure 2), this will make our GA loop stuck in the same loop everytime as this fitness is rewarding coins too much. Authors in (section 6.3 of [8]) also mention that this function is prone to stuck in local maximas and where the different objectives will have to be scaled properly in the fitness function in order for it to have any effect. And hence we aim to propose a robust and efficient fitness function.

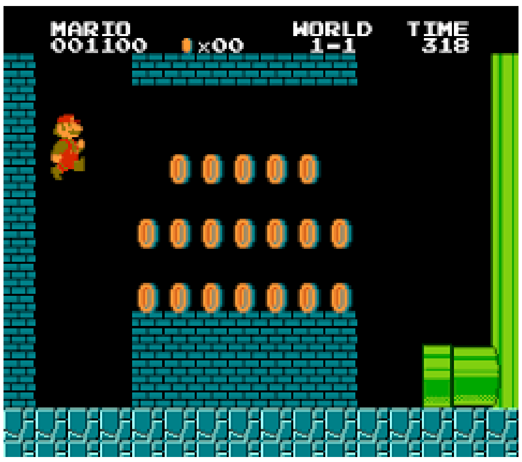


Figure 2: Snapshot from World 1 Level 1

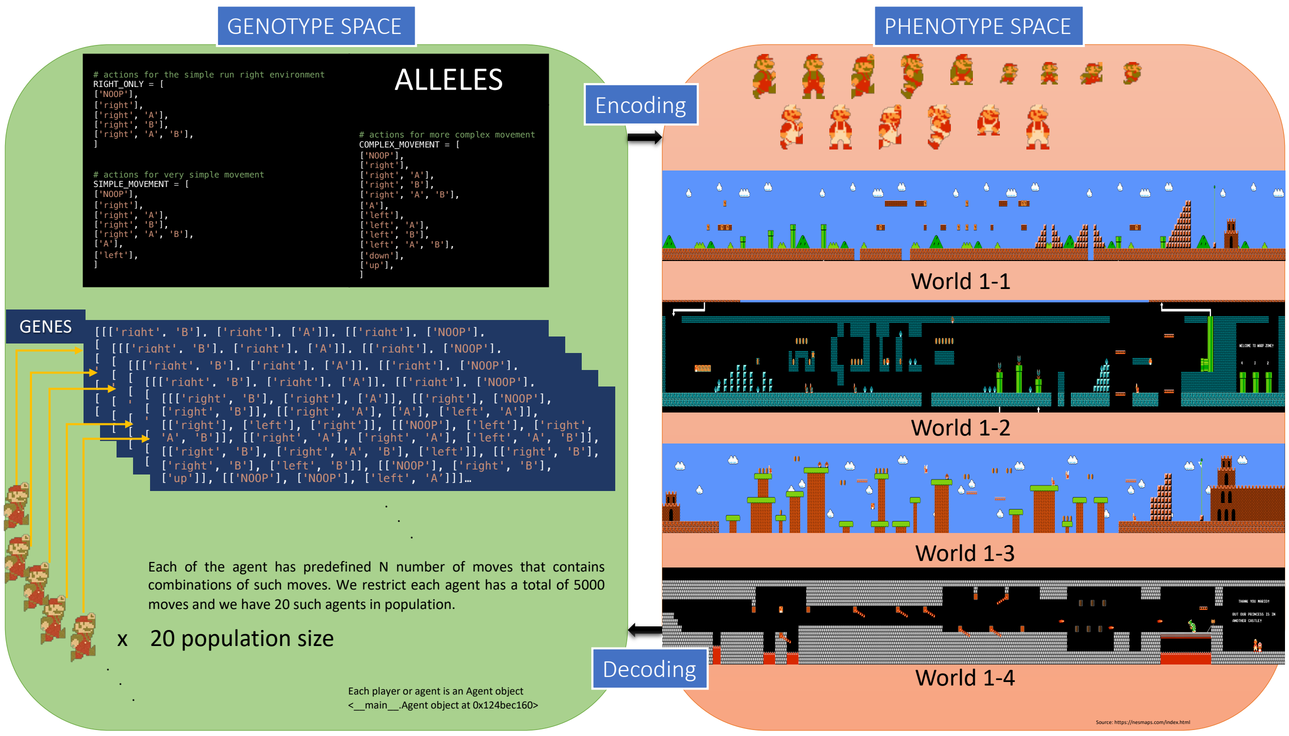
NE focuses on evolving artificial neural networks (ANNs) as controllers for Mario. It begins with a population of ANNs with random weights. These networks control Mario's actions and are evaluated based on their ability to help Mario achieve goals like coin collection and obstacle avoidance. Successful ANNs are preserved and mutated, gradually improving Mario's decision-making abilities through neural network evolution. NE adapts Mario's behavior to various challenges [2, 3, 7], making it a valuable tool for optimizing gameplay. In the literature none of the work actually focuses on constrained SMB environment. All of the recent work focuses a lot on how Mario as agent can perceive more elements from environment instead focusing on finishing levels with maximum coin collection in less moves and shorter duration. Moreover, the architectural inefficiency makes it under-performing as previous works are exploring multiple SMB agents that can finish game which increase their runtime of the algorithm. We claim that there is no need to converge for multiple agents, instead we only need one agent that can finish SMB environment level. Authors in [3] also mention that their limitation is lack of high processing power that would lead also to conduction of more experiments and probably better results. And as part of future work they also mention an approach with a harder version of the game which consists of more constraints can be developed on which our research work in this paper thrives.

We also study deep learning reward based approaches, however compaing MarioGA and MarioNE with such approaches is not scope of this paper. RL methodologies, such as Q-learning and Deep Q-Networks (DQNs), have garnered substantial attention due to their data-driven approach, enabling Mario to adapt and learn optimal strategies through interactions with the game environment. These RL-based methods equip Mario with the ability to make decisions based on rewards and penalties, which proves highly effective in navigating dynamic and evolving in-game scenarios. This adaptability, coupled with the capacity to discover novel gameplay strategies over time, positions RL as a vital component in the quest to optimize Mario's performance in SMB [9].

While these methods have shown significant promise in enhancing SMB's performance within constrained game environments, however, do not achieve optimal solutions through evolutionary and neural network-based strategies. Our specific constrained and objective is to collect as many coins with more distance traveled while in as short gameplay. We envision that these constraints will help our agent to converge faster and should result a robust SMB agent.

Proposed Methodology of MarioGA

In this proposed methodology, we elucidate how individuals, forming a population collectively, are represented. Each individual possesses a fitness or evaluation criteria, and various variation operators, including crossover and mutation, are employed. Selection is driven by higher fitness levels, portraying the principles of "survival of the fittest" and "mating of the fittest," ultimately optimizing



```
# actions for the simple run right environment
RIGHT_ONLY = [
['NOOP'],
['right'],
['right', 'A'],
['right', 'B'],
['right', 'A', 'B'],
]


# actions for very simple movement
SIMPLE_MOVEMENT = [
['NOOP'],
['right'],
['right', 'A'],
['right', 'B'],
['right', 'A', 'B'],
['A'],
['left'],
]
```

```
# actions for more complex movement
COMPLEX_MOVEMENT = [
['NOOP'],
['right'],
['right', 'A'],
['right', 'B'],
['right', 'A', 'B'],
['A'],
['left'],
['left', 'A'],
['left', 'B'],
['left', 'A', 'B'],
['down'],
['up'],
]
```

```
[[['right', 'B'], ['right'], ['A']], [['right'], ['NOOP'],
['right', 'B']], [['right', 'A'], ['A'], ['left', 'A']],
[['right'], ['left'], ['right']], [['NOOP'], ['left'], ['right',
'A', 'B']], [['right', 'A'], ['right', 'A'], ['left', 'A', 'B']],
[['right', 'B'], ['right', 'A', 'B'], ['left']], [['right', 'B'],
['right', 'B'], ['left', 'B']], [['NOOP'], ['right', 'B'],
['up']], [['NOOP'], ['NOOP'], ['left', 'A’]]]…
```

Each of the agent has predefined N number of moves that contains combinations of such moves. We restrict each agent has a total of 5000 moves and we have 20 such agents in population.

Each player or agent is an Agent object
<__main__.Agent object at 0x124bec160>

Figure 3: Individual's representation of MarioGA

on a fitness landscape. We also undertake the consideration of novelty vs quality tradeoff. In this tradeoff, two contrasting dynamics are at play: one aims to boost population variety through genetic mechanisms such as mutation and recombination, promoting the exploration of new possibilities, while the other strives to diminish population variety through selection processes that encompass both parental selection and survivor selection, with the primary goal of enhancing overall quality by exploitation. Additionally, literature discusses Binary, Integer, Real-Valued or Floating-Point, Permutation, and Tree representations for Genetic Operators in [5]. However, not necessarily the best ones for some applications.

Representation
One very important decision to make in beginning is for representation. As discussed concepts based on phenotype (object) and genotype (code) earlier in introduction, it is wise to note that representation provides a scope for perturbing and modifications using variation operators. We represent MarioGA agents in genotype space using several variables. In our case, the representation takes the form of a series of discrete actions that Mario can perform during the game, including jumping, moving left or right, and using power-ups. Each individual within the population is represented by a unique sequence of these actions, effectively defining its genotype. These sequences serve as the digital DNA of the agents and determine their behavior in the game world. This representation implies two essential mappings: encoding and decoding. Encoding translates the desired in-game behaviors (phenotype) into a corresponding sequence of actions (genotype), which is not necessarily one-to-one. Conversely, decoding translates the genotype back into the in-game behaviors, ensuring a one-to-one correspondence between the genotype and phenotype. These mappings are essential for translating between the genetic encoding and the agent's actual behavior. Furthermore, the representation involves the concept of "genes" and "alleles." Genes correspond to specific elements within the action sequences, while alleles represent the different possible values or actions that a gene can take. For instance, a gene associated with Mario's jumping behavior may have two possible alleles: "UP" and "RIGHT" In this genetic representation, every feasible solution is encoded within the genotype space, allowing the genetic algorithm to evolve and optimize Mario's gameplay strategies over successive generations.

To be specific about methodology, Agents are initialized with a sequence of moves that serve as their genetic representation as shown in Figure 3. This representation is encapsulated within the *self.moves* attribute of the *Agent* class, defining the agent's behavior in the game. The *randomize_moves*() method in the *Agent* generates a random sequence of actions to initialize the agent's moves. This is an essential step in creating the genetic representation for a new agent. During gameplay, the agent's actions (genetic representation) are executed and recorded. This recorded sequence of moves represents how the agent performs in the game environment (phenotype). When creating a population of agents, their genetic representations (sequences of moves) are generated using randomization. This initial representation sets the foundation for evolution in subsequent generations.

Evaluation or Fitness Function
In our case, the fitness function for an agent is defined as follows:

$$f(\text{agent}) = \text{CR} \cdot \text{CC} + \text{DR} \cdot \text{D} - \text{TP} \cdot (\text{MT} - \text{TL})$$

Where:

$$\begin{aligned} \text{CR} &: \text{Coin Reward} \\ \text{CC} &: \text{Collected Coins} \\ \text{DR} &: \text{Distance Reward} \\ \text{D} &: \text{Distance Traveled} \\ \text{TP} &: \text{Time Penalty} \\ \text{MT} &: \text{Max Time Allowed} \\ \text{TL} &: \text{Time Left} \end{aligned}$$

The fitness function employed in the MarioGA methodology serves as a critical component for evaluating the performance of each agent in playing the Super Mario Bros game. It calculates an agent's fitness score based on several key factors. Firstly, it rewards the agent for collecting coins, where the reward for each collected coin is denoted as CR (Coin Reward), and the actual number of coins collected by the agent during gameplay is represented as CC (Collected Coins). Secondly, it encourages the agent to travel a greater distance within the game, with DR (Distance Reward) as the reward factor and D (Distance Traveled) as the agent's recorded distance. Lastly, the fitness function introduces a penalty term to discourage agents from consuming excessive time during gameplay, where TP (Time Penalty) represents the penalty factor, MT (Max Time Allowed) denotes the maximum time allowed for completing the game level, and TL (Time Left) indicates the remaining time available to the agent. The fitness function balances these factors to assess an agent's overall performance, with the aim of evolving agents over generations to maximize their fitness scores and game-playing capabilities.

Selection Criteria
Our methodology for MarioGA uses a tournament selection mechanism as part of its genetic algorithm. In the $play_generation$ function, the code selects the elite agent (the one with the highest fitness) as a parent without making any changes to its moves. Then, for the rest of the population, it creates new agents through crossover and mutation. This approach is often referred to as "tournament selection" because it pits individuals against each other in small tournaments to determine which individuals will be selected as parents for the next generation. The tournament selection process helps maintain diversity within the population while favoring individuals with higher fitness. To implement tournament selection (as displayed in Figure 4) in a MarioGA, we follow the algorithmic description below:

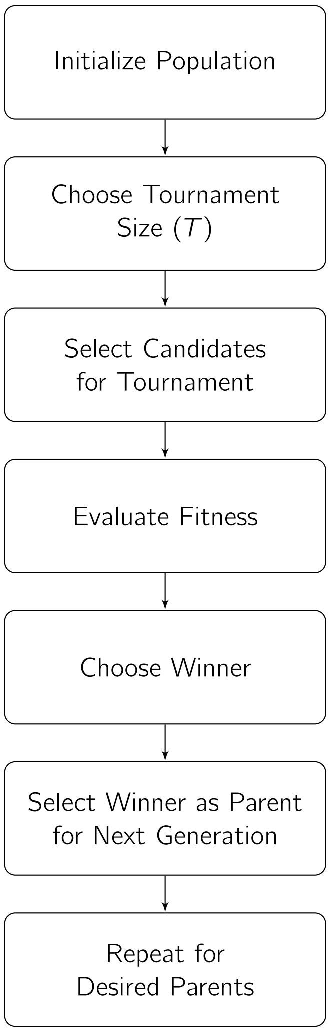

Initialize Population
Choose Tournament Size (T)
Select Candidates for Tournament
Evaluate Fitness
Choose Winner
Select Winner as Parent for Next Generation
Repeat for Desired Parents

Figure 4: Tournament Selection Process

1. Initialize a population P with individuals.
2. Choose a tournament size T (the number of individuals competing in each tournament).
3. Repeat the following steps for each parent you need to select:
   (a) Randomly select T individuals (candidates) from the current population P.
   (b) Calculate the fitness of each candidate.
   (c) Choose the candidate with the highest fitness as the winner.
   (d) The winner is selected as a parent for the next generation.
4. Repeat step 3 for the desired number of parents.

Recombination or Crossover

In the context of MarioGA, the crossover (recombination) operator is used to create new agent individuals (offspring) by merging genetic information from two parent agents. Each agent is represented by a sequence of moves that determine its gameplay strategy. The crossover operator mimics genetic recombination in natural reproduction, allowing promising strategies from different agents to be combined to potentially produce more successful offspring. Example shown in Figure 5 displays how it is working in MarioGA.

The crossover type used is One-Point Crossover in MarioGA which is described as:

- Select two parent agents, each represented by a sequence of moves.
- Choose a crossover point within the move sequences.
- Exchange move sequences to create two new agent offspring:
    - Offspring 1: Combines moves from Parent 1 before the crossover point with moves from Parent 2 after the crossover point.
    - Offspring 2: Combines moves from Parent 2 before the crossover point with moves from Parent 1 after the crossover point.
- One-point crossover in this context enables the genetic algorithm to explore different gameplay strategies by mixing and matching moves from successful parent agents.

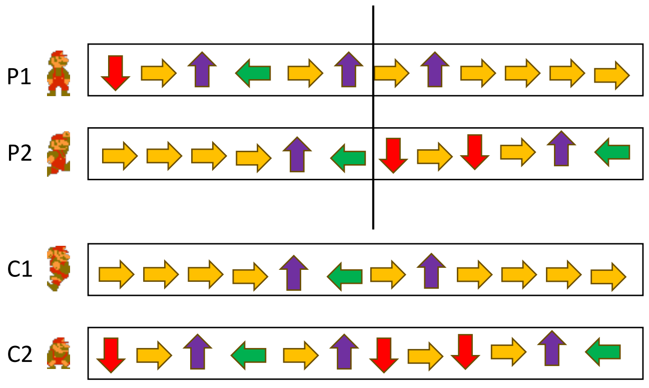


Figure 5: Single point crossover at cross-over point at 50% of the total size of an Individual sequence of moves.

Mutation

Mutation is a fundamental operator in genetic algorithms (GAs) that introduces randomness and diversity into the population of individuals (genomes). It plays a critical role in maintaining genetic diversity within the population, preventing premature convergence to sub-optimal solutions, and enabling the exploration of new areas of the search space. In MarioGA, mutation is implemented as follows:

1) Mutation Rate (*mutation_rate*): A mutation rate is defined at the beginning, indicating the probability of a move being mutated. The mutation rate is set to 0.8 (80%) for our case, which means there is an 80% chance that a move will be mutated. 2) *mutate_moves* Function: The *mutate_moves* function is responsible for introducing mutations into an agent's moves. It selects a starting index in the list of moves based on the percentage of moves that are mutable (*moves_mutable*). Then, for each move from that index to the end of the list, it randomly decides whether to replace the move with a random action sampled from the environment's action space, based on the mutation rate. 3) Mutation of Moves: Mutation occurs at the level of individual moves within an agent's sequence of actions. The algorithm iterates through the list of moves, and for each move, it checks if a random number is less than the mutation rate. If it is, the move is replaced with a random action.

Hence, mutation is performed by randomly replacing a subset of an agent's moves with new, randomly selected actions from the environment's action space, and this mutation is a type of random point mutation.

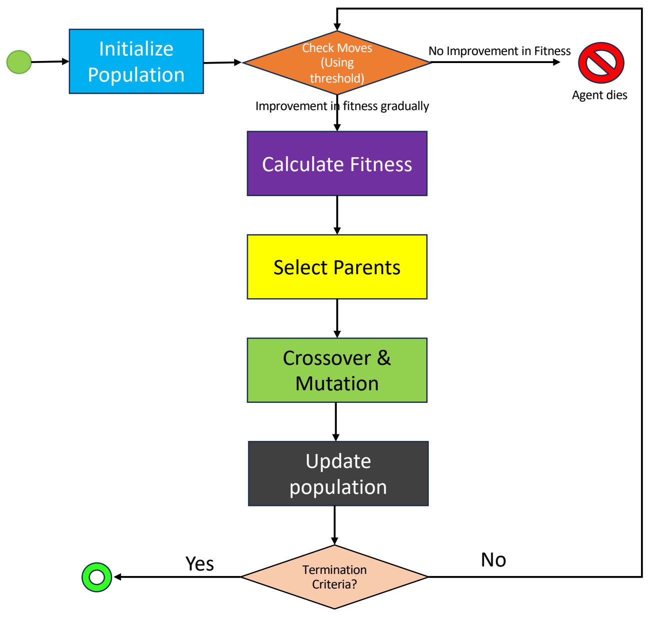


Figure 6: Full working pipeline of the proposed methodology for MarioGA.

Termination and Working Algorithm

The full working pipeline of proposed methodology is shown in Figure 6. Additionally, the working algorithm is present in Algorithm 1. The termination criteria are based on the following conditions:

1) No Improvement in Fitness: If an agent fails to show any improvement in its fitness over a certain number of moves (specified by *moves_to_check*), it

is terminated. This condition ensures that agents evolve efficiently and do not waste computational resources on stagnant individuals.

2) Generation Limit: The code runs for a predefined number of generations ($generation\_amount$) to control the overall duration of the evolutionary process. Once this limit is reached, the evolution process terminates, and the best-performing agent is selected as the final result.

The combination of these criteria helps to ensure that the genetic algorithm converges towards improved solutions and avoids unnecessary computation when it reaches a point where further evolution is unlikely to yield better results.

Algorithm 1: Mario Genetic Algorithm (MarioGA)

```
Data: Initial population of Mario agents
Result: Evolved population of Mario agents
Input : Maximum generations, fitness threshold,
        mutation rate
Output: Optimal Mario agent

BEGIN
INITIALIZE population with random Mario agents;
EVALUATE fitness for each Mario agent;
repeat
    SELECT parents based on fitness;
    RECOMBINE pairs of parents to create offspring;
    MUTATE the moves of resulting offspring;
    EVALUATE fitness for the new generation of Mario agents;
    SELECT individuals for the next generation;
until TERMINATION CONDITION is satisfied;
END
```

Proposed Methodology of MarioNE

MarioNE leverages NE to evolve increasingly proficient Mario agents over generations. This section explores the key components and definitions of MarioNE.

Neuroevolution (NE)

Neuroevolution is a subfield of artificial intelligence and machine learning that applies evolutionary algorithms to optimize artificial neural networks (ANNs) for solving complex tasks. In the context of MarioNE, NE is used to evolve neural network-based Mario agents.

Neuro Evolution Operators

1) Population: The population consists of a group of Mario agents, each represented by a unique neural network. During NE, multiple agents coexist within the population, and their performance is evaluated to select the most fit individuals. 2) Fitness Function: The fitness function quantifies the performance of a Mario agent in the game. It assigns a numerical score based on how well the agent accomplishes specific objectives, such as collecting coins, avoiding obstacles, and reaching the level's end. 3) Selection: The selection process in NE involves choosing the fittest Mario agents from the population to become parents for the next generation. Agents with higher fitness scores have a greater chance of being selected. 4) Crossover: Crossover is a genetic operator where pairs of parent Mario agents exchange genetic material (neural network parameters) to create offspring. It introduces diversity in the neural networks of the next generation. 5) Mutation: Mutation is another genetic operator that introduces small random changes to the neural networks of Mario agents. It helps explore new behaviors and prevents the population from converging prematurely. 6) Termination Criteria: Termination criteria are conditions that determine when the NE process should stop. Common termination criteria include reaching a predefined fitness threshold, a maximum number of generations, or a fixed computational budget. 7) Evolutionary Progress: The evolutionary progress in MarioNE is measured by observing how the fitness of the population changes across generations. As the NE process continues, Mario agents evolve to become more proficient at playing the game.

MarioNE Methodology

In this section, we detail the methodology for employing Neuroevolution (NE) to evolve intelligent agents for playing Super Mario Bros. MarioNE aims to optimize neural network-based agents to achieve better performance in the game while adhering to the fitness function introduced in MarioGA. The following steps outline the MarioNE methodology:

Neuroevolution Framework

To implement MarioNE, we utilize a neuroevolution framework that provides tools for evolving neural network-based agents. We choose the NEAT (NeuroEvolution of Augmenting Topologies) algorithm due to its ability to evolve both neural network weights and structures. NEAT maintains a population of neural networks with varying complexities, allowing for the exploration of different strategies.

Neural Network Architecture

We design a neural network architecture that serves as the brain of our Mario agents. The neural network takes game state information as input and produces actions as output. The architecture includes layers for processing input data, hidden layers for learning complex patterns, and output layers for mapping to Mario's actions. We employ activation functions like the rectified linear unit (ReLU) to introduce non-linearity.

Fitness Function

The fitness function in MarioNE mirrors the one used in MarioGA to align with the objectives of collecting as

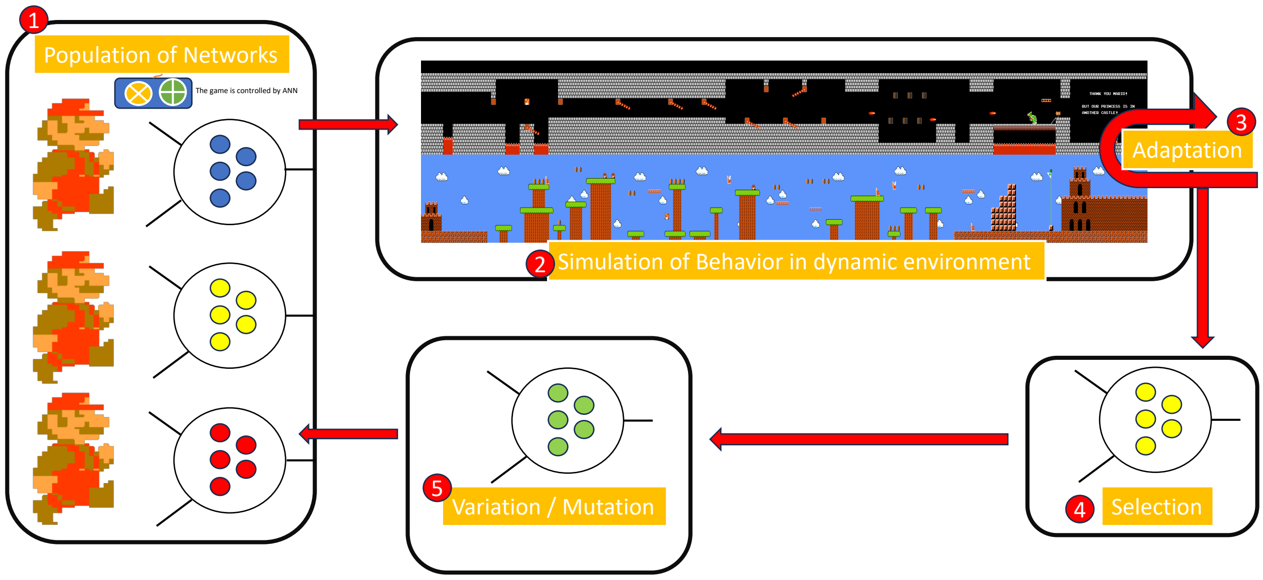

1
Population of Networks
The game is controlled by ANN
2
Simulation of Behavior in dynamic environment
3
Adaptation
4
Selection
5
Variation / Mutation

Figure 7: Full working pipeline of the proposed methodology for MarioNE.

many coins as possible, traveling a large distance, and minimizing time spent in the game. The fitness function is defined as follows:

$$\text{Fitness} = \alpha \cdot \text{Coins} + \beta \cdot \text{Distance} - \gamma \cdot \text{Time} \qquad (1)$$

Where:

- α, β, and γ are trade-off parameters.
- Coins represent the number of collected coins.
- Distance is the progress made through the level.
- Time is the time taken to complete the level, with a penalty.

The trade-off parameters allow us to balance the importance of each objective in the fitness function.

Neuroevolution Process

The core of MarioNE is the neuroevolution process, which optimizes Mario agents' neural networks over generations. We employ the following steps:

1. Initialization: Initialize a population of Mario agents with randomly generated neural networks.
2. Evaluation: Evaluate the fitness of each agent by running them through Super Mario Bros.
3. Selection: Select agents for reproduction based on their fitness scores. Higher fitness leads to a higher chance of selection.
4. Crossover: Create offspring by combining the neural networks of selected parents. Our implementation (discussed in later sections of the paper) also facilitates crossover with the preservation of innovation history.
5. Mutation: Introduce small random changes to the offspring's neural networks to maintain diversity.
6. Evaluation (Next Generation): Evaluate the fitness of the new generation of agents in the game environment.
7. Termination: Repeat the process for multiple generations or until a termination criterion (e.g., No improvement in fitness) is met.

Through this iterative process, Mario agents evolve to exhibit increasingly intelligent gameplay behaviors. A simple pipeline of the working of NeuroEvolution is shown in Figure 7.

Implementation details of MarioGA

We implement MarioGA using python and supporting libraries of Gym-OpenAI. Following are critical details of our implementation.

Agent Class

The code defines an `Agent` class representing an individual agent. Each agent possesses the following attributes:

- `fitness`: A measure of an agent's performance.
- `moves`: A sequence of actions representing the agent's gameplay.
- `moves_used`: The count of moves executed by the agent.

Genetic Algorithm Components

- `create_population()`: This function generates a population of agents. It creates agents with random moves.

Table 2: Genetic Operators in MarioGA

Genetic Operator	**Operator Description**	**Operator Value**
Representation	List or Sequence of actions and moves	['right', 'A', 'B', 'left','down','up']
Crossover	One-Point Crossover	50%
Mutation	Flip and Change (adaptive mutation with a fraction for fittest)	0.01±0.005
Population	Collection of all individuals	20
Generation Size	Number of iterations to evolve agents	1000
Moves Amount	Number of all moved that each agent posses before gameplay	5000
Moves to Check	Number of moves that do not lead to an increase in fitness, (agent stuck in suboptimum)	30
Moves Mutable	This value determines how much percentage of moves are going to be mutated or perturbed	80%
Coin Reward (+)	This gets multiplied with the amount of coins collected by Mario in gameplay	10
Distance Reward (+)	This gets multiplied with the distance traveled by Mario in gameplay	0.1
Time Penalty (-)	This gets multiplied with the total time quantum in gameplay	0.8
Elitism	We preserve at least some best agent from the last generation	1
Number of Off-spring	Children produced after crossover or recombination operator	2
Initialisation	How are we initializing the agents or agent sequence of moves	Random
Termination	We use two types of termination criteria	'generation size', 'moves to check'

- `check_fitness()`: This function calculates an agent's fitness based on various factors, including coins collected, distance traveled, and time survived. It also implements a mechanism for terminating agents with no fitness progress.
- `mutate_moves()`: Agents undergo random mutations in their move sequences. The mutation rate and the portion of moves that can be mutated are configurable.
- `play_generation()`: This function simulates a generation of agents' gameplay. It selects the fittest agent, preserves it, and applies crossover and mutation to the rest of the population. The mutation rate is adjusted based on fitness progress.
- `record_player()`: It records the gameplay of an agent, stores images, and calculates its fitness. The best agent's gameplay is captured for animation.

Main Loop

The main loop iterates through generations and evolves the population of agents using the genetic algorithm. The key steps include:

1. Creating a new population of agents.
2. Simulating the generation by playing agents, calculating fitness, and evolving the population.
3. Saving the best agent's gameplay as animations in a dedicated folder.
4. Adjusting mutation rates based on fitness progress.
5. Continuously monitoring and recording the highest fitness achieved in each generation.

Parameters

The code allows various parameters to be configured, such as mutation rate, population size, and fitness components (coin reward, distance reward, time penalty). These parameters enable fine-tuning of the genetic algorithm's behavior.

Custom Starting Agent

The `custom_starting_agent()` function enables the use of a predefined agent to start from a specific point in the training process.

Overall, our Python code implements a comprehensive genetic algorithm (MarioGA) for training an agent to play SMB. It includes various functions for agent management, fitness calculation, and population evolution. The code allows for extensive customization and records the best agent's gameplay as animations for analysis. For any code related or explanation of the variables, we refer code. Moreover, we present a tabular summary in Table **??** of the implementation of MarioGA with description of the GA.

Table 3: Genetic Operators in MarioNE

Genetic Operator	Operator Description	Operator Value
Representation	Neural Network Architecture	Dynamic, Evolving
Initialization	Random Initialization of Neural Networks	Random Weights
Fitness Evaluation	Assessment of Agent Performance	Coin Collection, Distance Traveled, Time Penalty
Population	Collection of Neural Networks	1 (Configurable)
Generations	Number of Iterations in Evolution	1000 (Configurable)
Mutation Rate	Probability of Weight Mutation	10% (Configurable)
Number of Actions	Available Actions for Mario	Variable (e.g., RIGHT_ONLY)
Crossover	Reproduction Through Genetic Recombination	One-Point Crossover
Selection	Choosing Top Performers for Reproduction	Top 20% (Configurable)
Termination	Criteria for Ending Evolution	Fitness Threshold, Maximum Generations

Implementation details of MarioNE

The Mario Neuro Evolution (MarioNE) framework leverages NeuroEvolution of Augmenting Topologies (NEAT) to train Mario agents within the context of Super Mario Bros. NEAT is a powerful neuroevolutionary algorithm that evolves neural networks with dynamic architectures, making it well-suited for complex tasks like mastering video games. MarioNE's implementation involves several key components and stages which are also summarised in Table 3, as outlined below:

Neural Network Architecture

The neural networks in MarioNE are fundamental to agent decision-making. Each neural network is instantiated with a random weight matrix, connecting the input layer (representing the game state) to the output layer (determining Mario's actions). NEAT allows dynamic construction of hidden layers, adding nodes and connections as the networks evolve over generations. This adaptability enables the agents to discover complex strategies through learned behaviors.

Initialization and Population

At the beginning of each NEAT-based evolutionary run, MarioNE initializes a population of neural networks. These networks collectively represent the evolving Mario agents. The population size is a configurable parameter, and typically, it comprises multiple neural networks with varying structures.

Fitness Evaluation

To assess the performance of each agent, a fitness function is employed. The fitness function is designed to optimize specific objectives while penalizing undesired behaviors. In the case of MarioNE, the fitness function incentivizes agents to collect coins, traverse longer distances, and complete levels within minimal time. Simultaneously, it penalizes extended gameplay time. The fitness evaluation process simulates Mario's gameplay using the neural network's decisions and records the achieved fitness scores.

NEAT Evolutionary Process

MarioNE employs the NEAT algorithm to evolve the neural networks over a predefined number of generations. During each generation, the following steps are executed:

1. Selection of Top Performers: The top-performing agents, typically the top 20% based on fitness scores, are selected for reproduction. This selective pressure ensures that the most promising agents propagate their genetic information to the next generation.

2. Crossover and Mutation: To maintain diversity and facilitate exploration, new agents are created through crossover and mutation operations. In the crossover step, two parent networks contribute to the genetic makeup of a child network, allowing for the exchange of advantageous traits. Mutation introduces variability into the child's neural network structure and connection weights. These operations are governed by NEAT-specific mechanisms.

3. Population Replacement: The newly created agents, including the top performers from the previous generation, replace the previous population, forming the next generation. This step ensures that genetic diversity is maintained while enabling further evolution.

Termination and Best Agent

MarioNE iterates through generations until a predefined termination condition is met, such as achieving a certain fitness threshold or reaching a specified maximum number of generations. The best-performing agent, based on the highest fitness achieved, is identified and considered the output of the evolutionary process.

MarioNE's implementation details encompass neural network encoding, initialization, fitness evaluation, selection, crossover, mutation, and termination management. By combining NEAT's flexibility with Super Mario Bros.' gameplay, MarioNE endeavors to evolve proficient Mario agents capable of navigating the game environment adeptly.

Results and Discussions

In this section we will discuss all results and noteworthy discussions for both the approaches. Firstly, results include worst, average, and best fitness plots performed with five-fold cross validation for each level. In simpler words, we provide fitness plot for both the algorithms that shows generation vs fitness of the MarioGA and MarioNE respectively for world 1 levels 1, 2, 3 and 4. In addition, we also perform same experiment 5 times for each level and average out the results to display in a single plot in order to prove that it was not the lucky case that MarioGA and MarioNE were able to finish levels. Secondly, we provide additional plots of generation vs the times Mario got stuck in local optimum (cases where there is no progress in fitness, which we discussed in Figure 2). Third, we plot datatset distribution or environment landscape, which in our case is SMB's levels such that it could display the local-optimums where agent could stuck. Fourth, we perform empirical comparisons of both the algorithms on the basis of tweaking various constants (α, β and γ) of Fitness Equation 1 and plot respective generation vs fitness plots. Fifth, we study the dynamics of the rewards and penalties (in different colors) of different levels within SMB gameplay. This bar plot will provide information of landmarks where it was able to increase the fitness sharply. Additionally, we provide a table where we provide statistics of the gameplay. This statistics include best agent's distance covered, time taken for the agent to finish level, damage taken, kills, coins collected, obstacle's killed or avoided, jumps taken, right move performed, left move performed, life (power ups) discovered and eaten, time taken to find best agent and number of mutations performed in order to reach this best agent. Finally, we discuss running performance of both the algorithms, associating the theoretical time complexity of the algorithms with their experimental performance and interpret all findings. In other words, we compare how the algorithms are expected to perform based on their theoretical analysis (i.e., the time complexity) with how they actually perform in the experiments.

Plots for MarioGA

In this subsection we particularly focus on plotting Genetic Algorithm experiments. We plot three figures for each world (We perform 5 experiments for each level of world one of the SMB gameplay). For details about plots we refer caption of the plot. For World 1 level 1, Figures 8, 9, 10 are shown. For World 1 level 2, Figures 11, 12, 13 are shown. For World 1 level 3, Figures 14, 15, 16 are shown. For World 1 level 4, Figures 17, 18, 19 are shown.

Conclusion

References

1. Kauten C. Super Mario Bros for OpenAI Gym. GitHub. 2018. Available from: https://github.com/Kautenja/gym-super-mario-bros
2. Mautschke O. Evolutionary algorithms for Controllers in Games. AI for Games - Oliver Mautschke 2019
3. Bytyçi E, Hulaj A, and Aliu D. Combination of Genetic Algorithm and Neural Network for Independent Game Playing. *2020 International Conference on Intelligent Data Science Technologies and Applications (IDSTA)*. IEEE. 2020 :106–9
4. Jain S. RAMario: Experimental Approach to Reptile Algorithm–Reinforcement Learning for Mario. arXiv preprint arXiv:2305.09655 2023
5. Eiben AE and Smith JE. Introduction to evolutionary computing. Springer, 2015
6. Floreano D, Dürr P, and Mattiussi C. Neuroevolution: from architectures to learning. Evolutionary intelligence 2008; 1:47–62
7. Risi S and Togelius J. Neuroevolution in games: State of the art and open challenges. IEEE Transactions on Computational Intelligence and AI in Games 2015; 9:25–41
8. Tønder LS and Olsen OP. Multi-Objective Neuroevolution in Super Mario Bros. MA thesis. Institutt for datateknikk og informasjonsvitenskap, 2013
9. Engelsvoll RN, Gammelsrød A, and Thoresen BS. Generating levels and playing Super Mario Bros. with deep reinforcement learning using various techniques for level generation and deep q-networks for playing. MA thesis. University of Agder, 2020
10. Andersen PA, Goodwin M, and Granmo OC. Deep RTS: a game environment for deep reinforcement learning in real-time strategy games. *2018 IEEE conference on computational intelligence and games (CIG)*. IEEE. 2018 :1–8

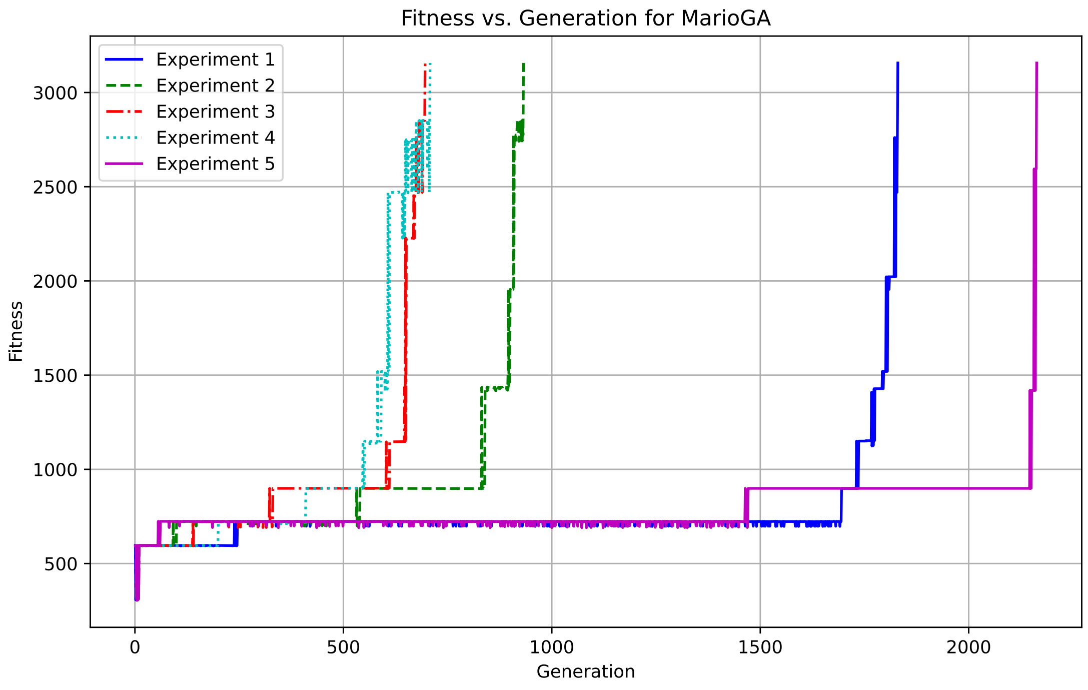


Figure 8: Fitness vs Generation plot for World 1 Level 1. Experiments 3 and 4 were able to converge faster without stuck in some local optimum while experiments 1 and 5 were stuck in local maximum for longer generations. This is where our Mario agent was dying repeatedly and there was no progress for longer generations.

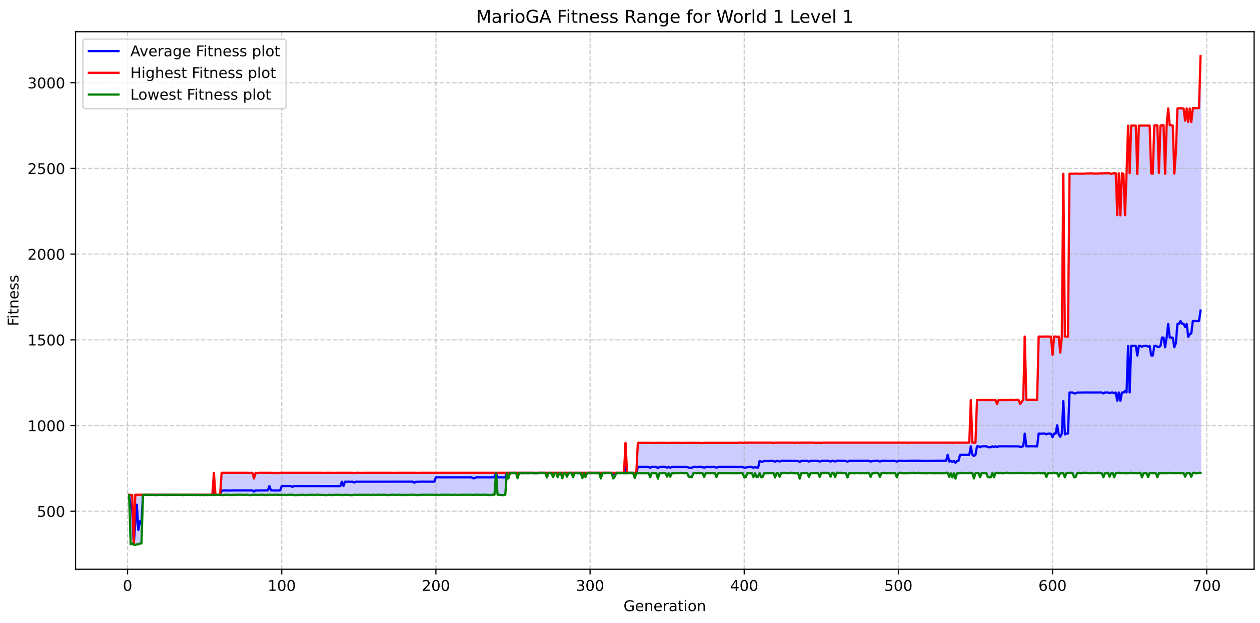


Figure 9: Highest, Average and Lowest fitness compared for 700 generations. This plot says that on average MarioGA was achieving sub-optimum fitness of ±1300 while if we look for highest fitness for 700 generations, it was able to converge with a fitness of ±3000

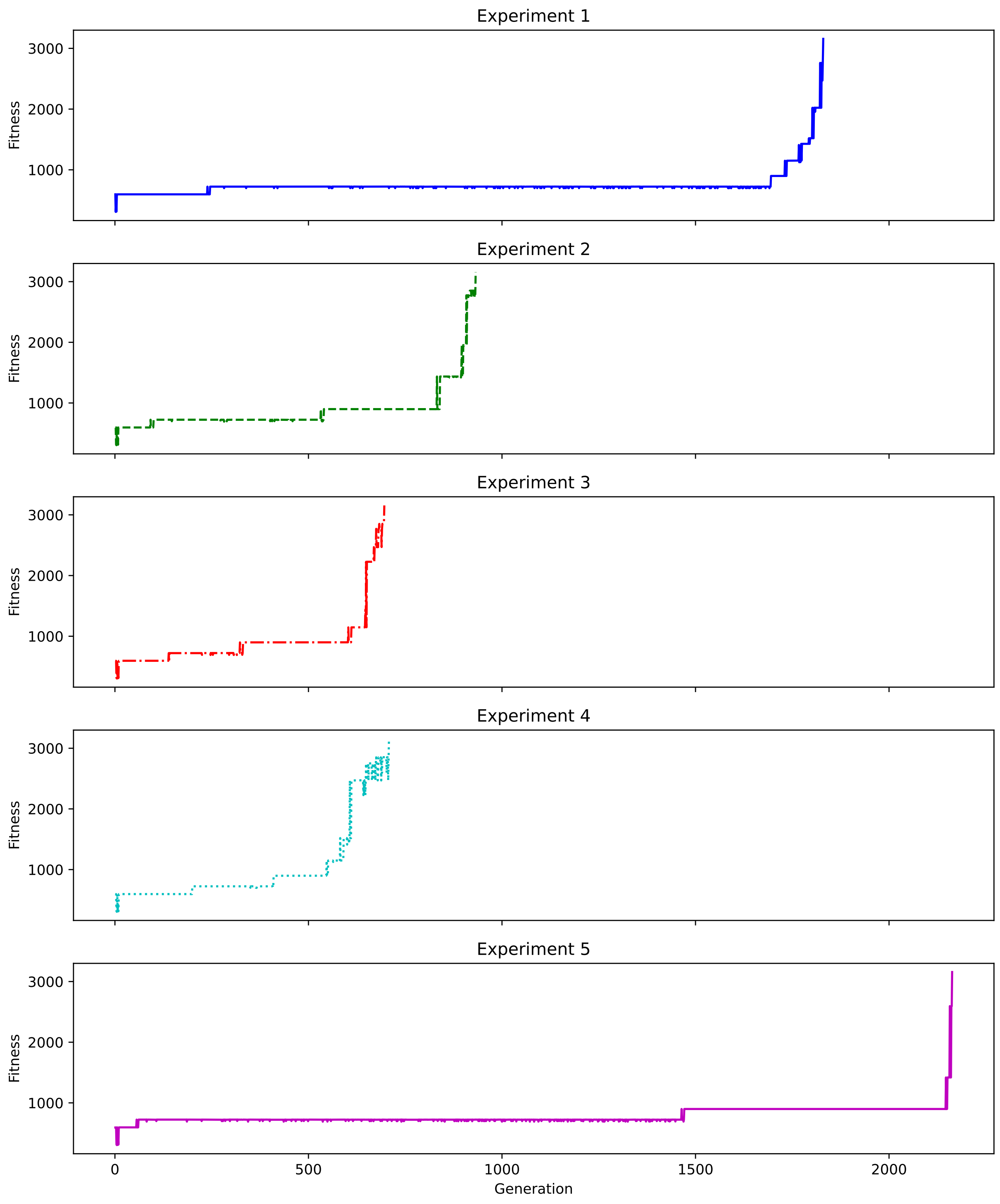


Figure 10: This plot shows five different experiments analysed separately. Overall all experiments were able to finish the level 1.

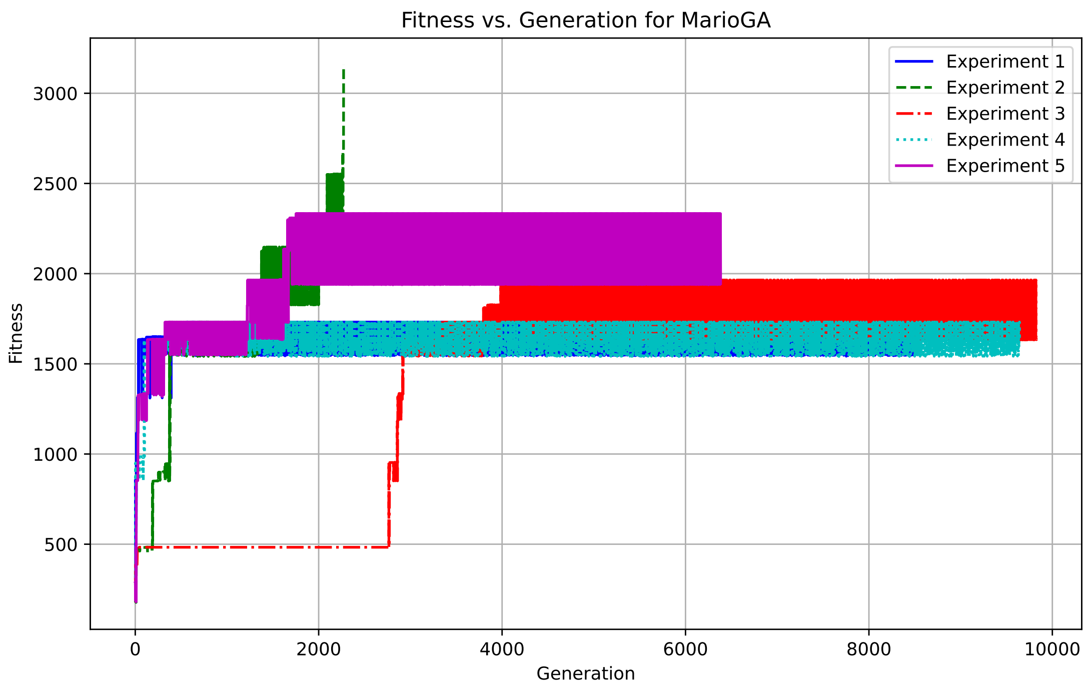


Figure 11: Fitness vs Generation plot for World 1 Level 2. Only Experiment 2 was able to finish the level in very short duration achieving highest fitness of ±3125, however rest of the experiments did not converge even up to 10000 generation size.

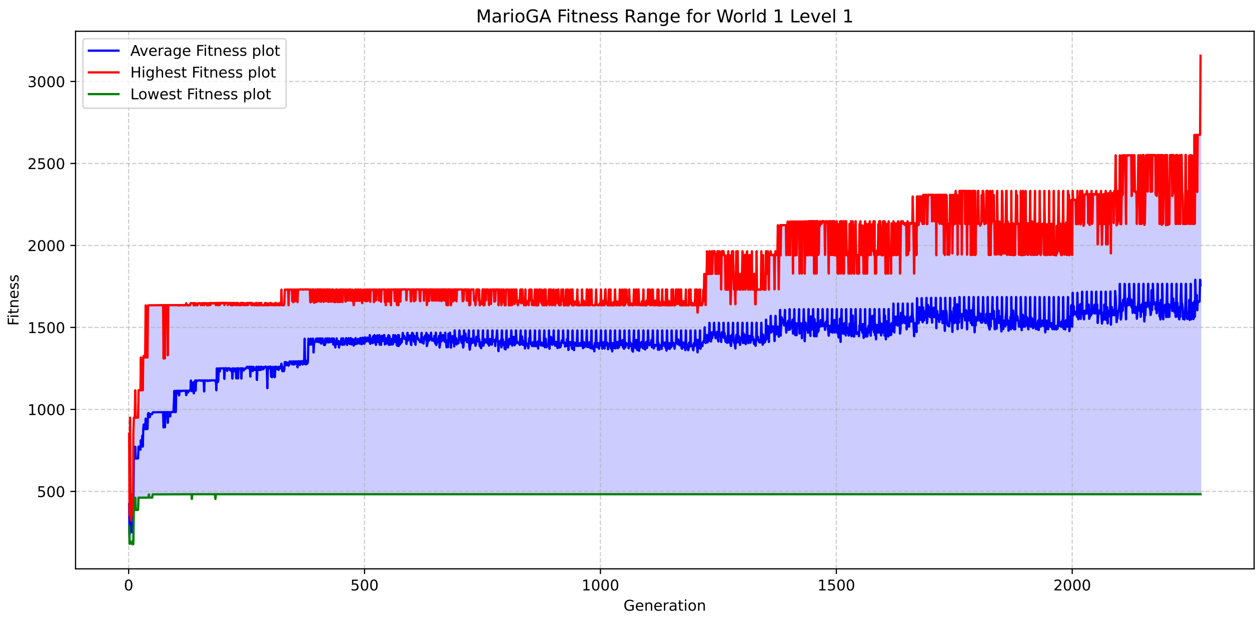


Figure 12: Highest, Average and Lowest fitness compared for 2500 generations. This plot says that on average MarioGA was achieving sub-optimum fitness of ±1700 while if we look for highest fitness for 2500+ generations, it was able to converge with a fitness of ±3125

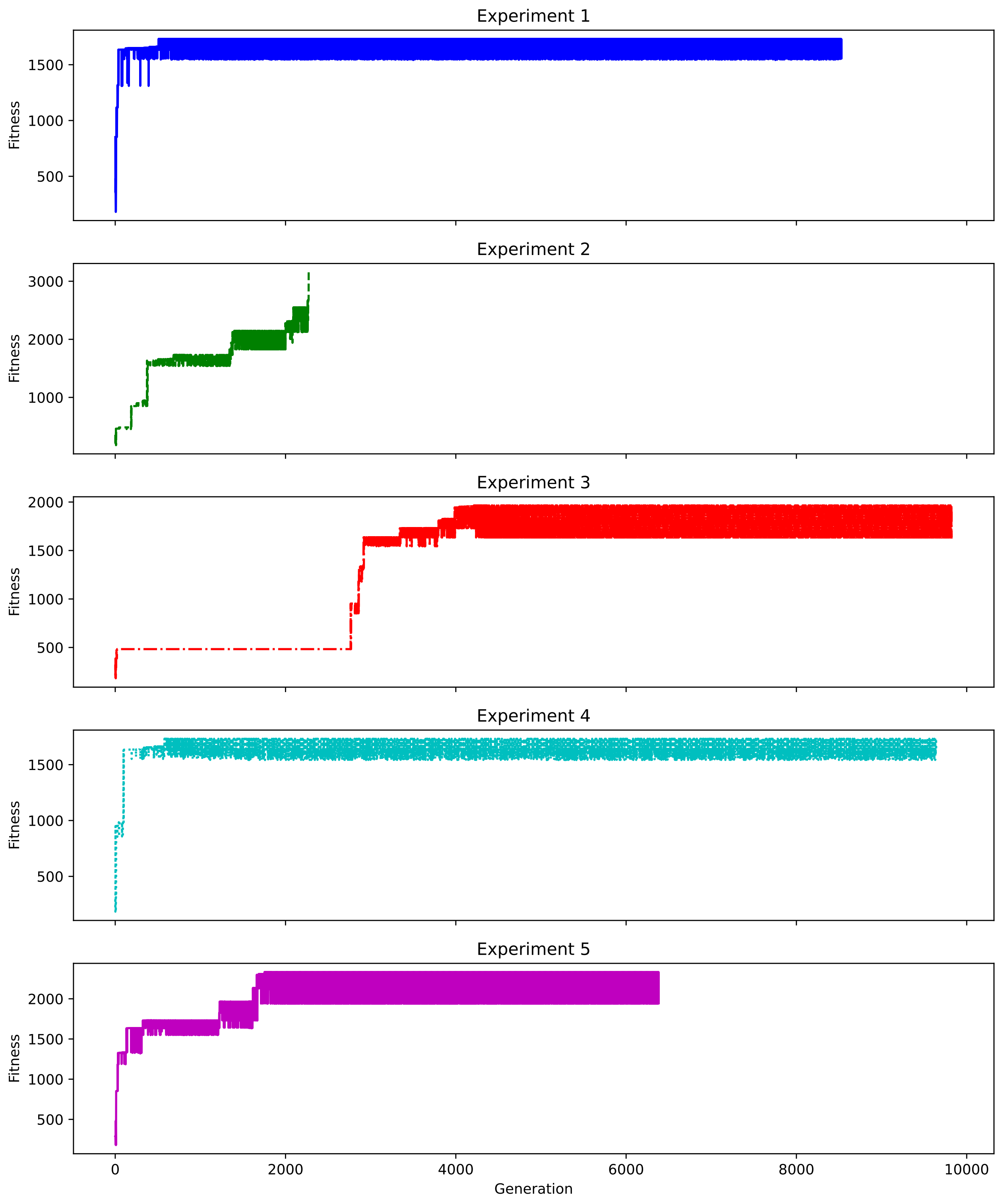


Figure 13: This plot shows five different experiments analysed separately. Overall only experiment 2 was able to finish the level 2.

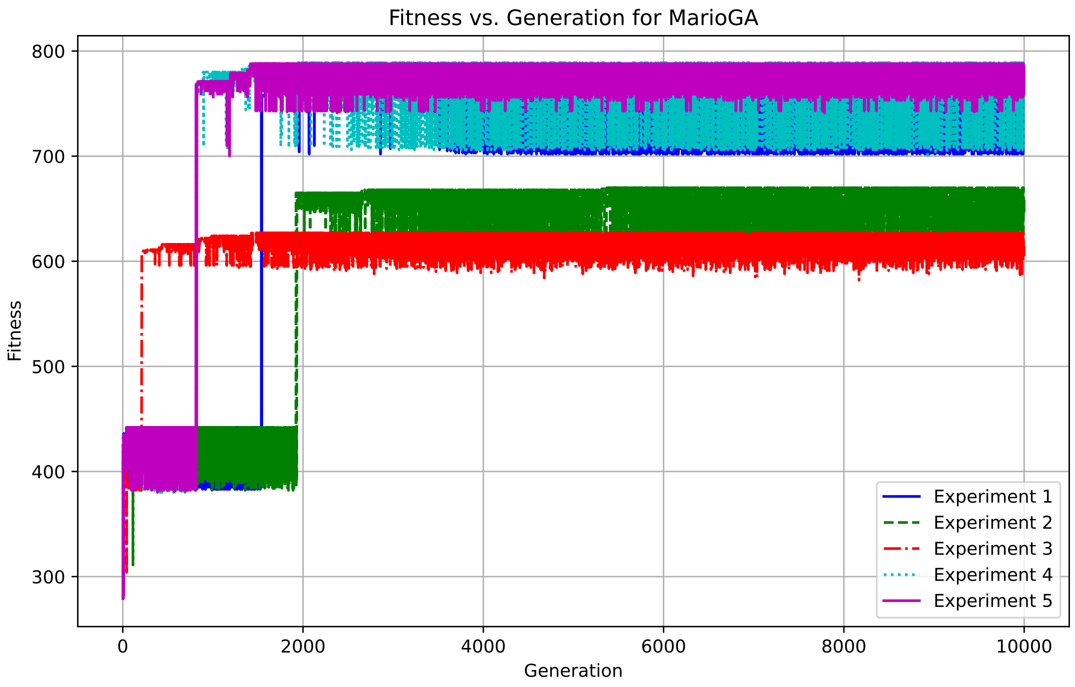


Figure 14: Fitness vs Generation plot for World 1 Level 3. No Experiment was able to finish the level 3, even after running for 10000 generations.

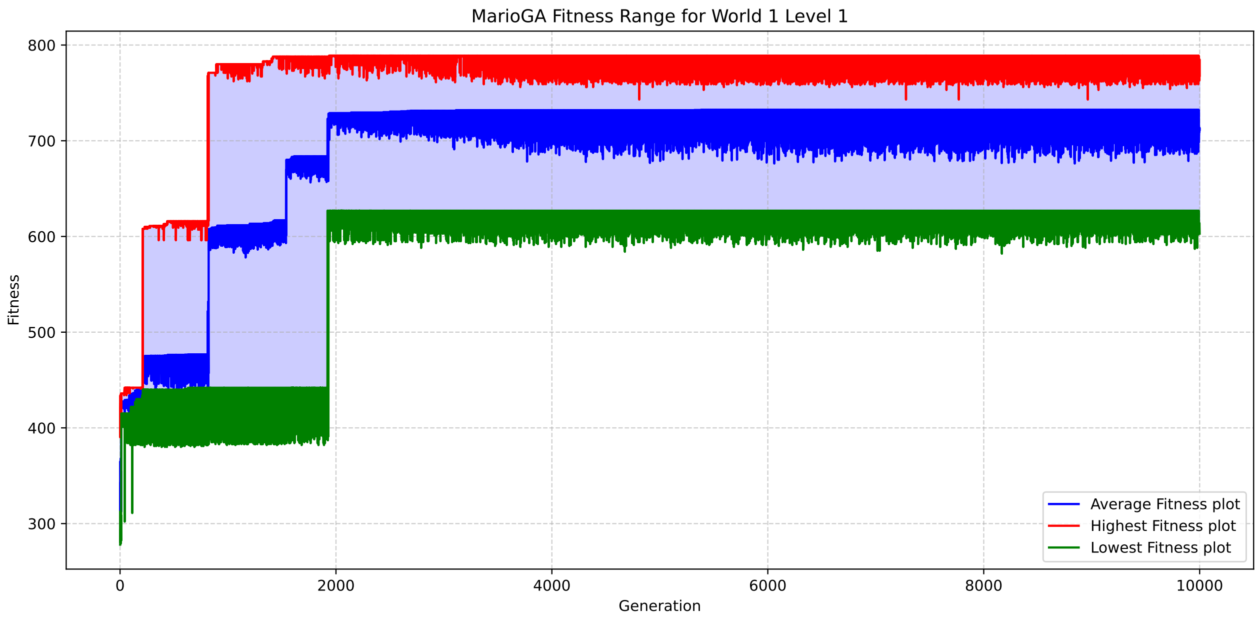


Figure 15: Highest, Average and Lowest fitness compared for 10000 generations. This plot says that on average MarioGA was achieving sub-optimum fitness of ±700 while if we look for highest fitness for 10000 generations, it was not able to converge with a fitness of ±800.

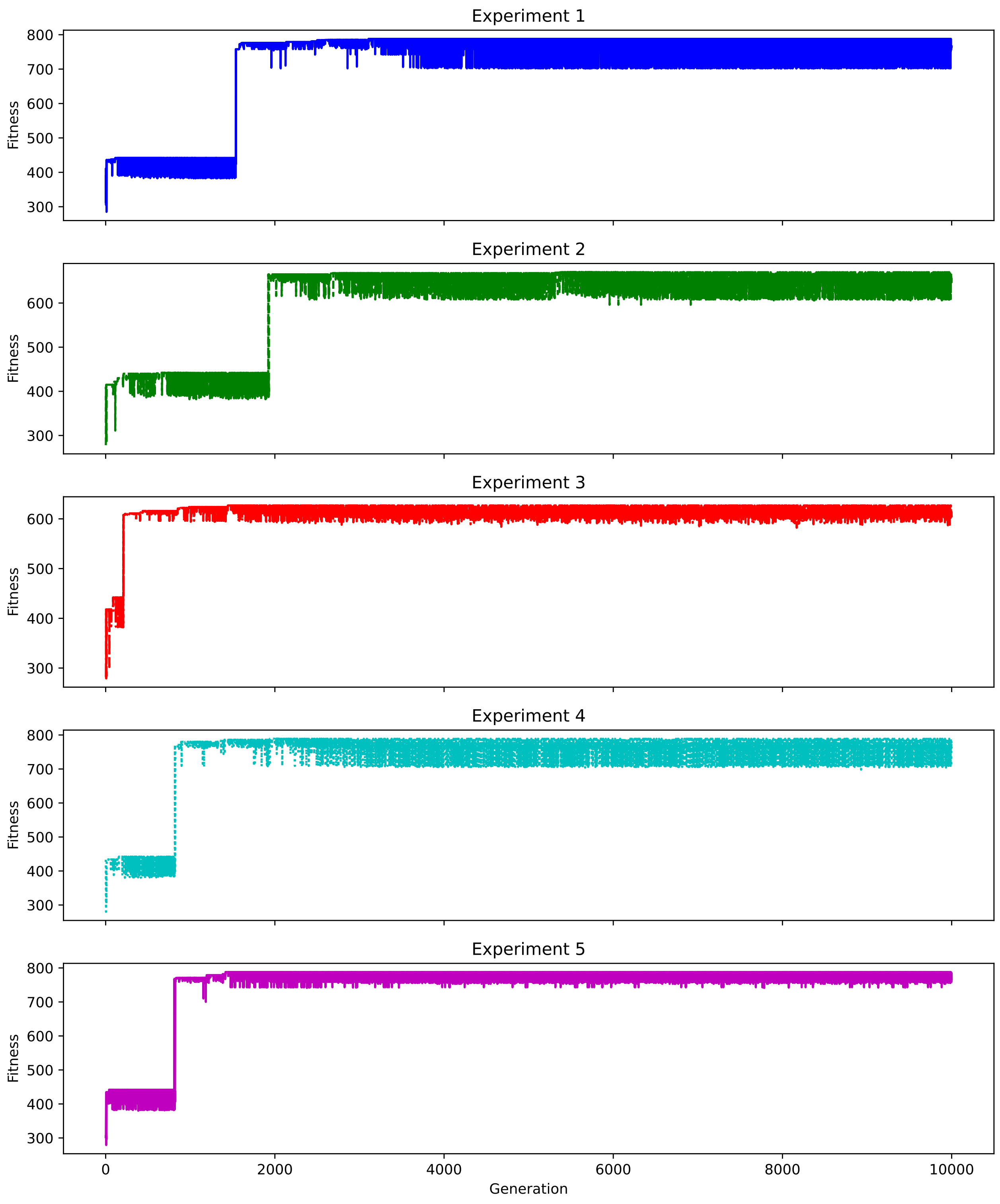


Figure 16: This plot shows five different experiments analysed separately. Overall no experiment was able to finish the level 3.

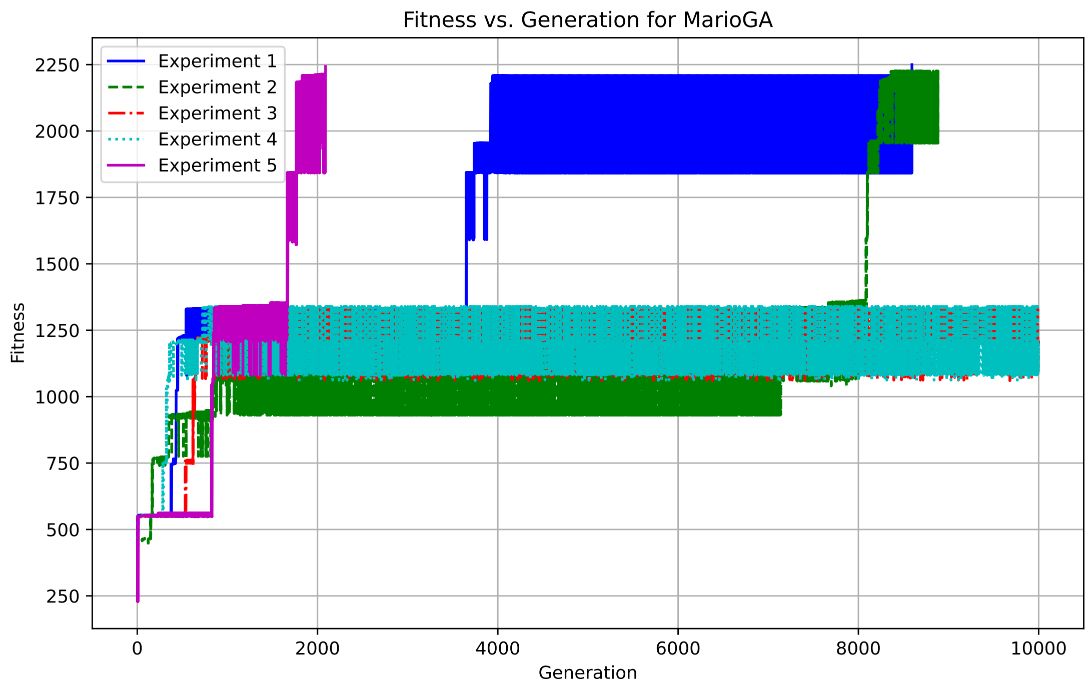


Figure 17: Fitness vs Generation plot for World 1 Level 4. Experiments 1, 2 and 5 were able to finish the gameplay with a fitness of ±2250. Experiment 1 was able to finish gameplay in 8000+ generations, Experiment 2 was able to finish gameplay in ±8200 generations and Experiment 5 was able to finish gameplay in 2200+ generations.

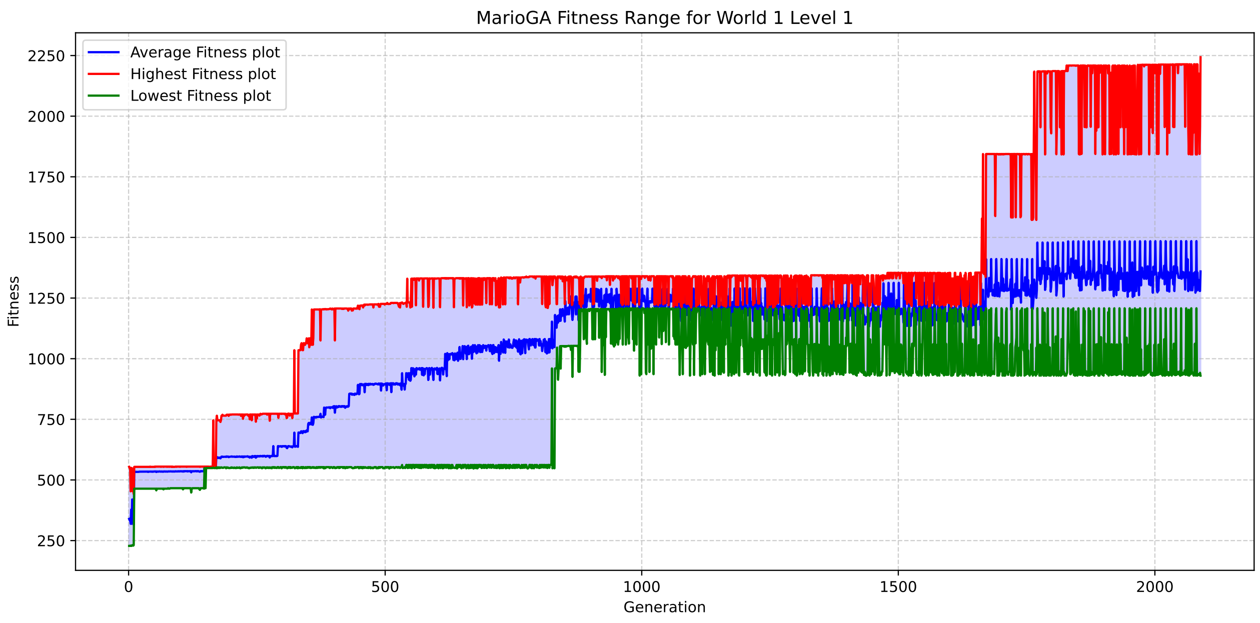


Figure 18: Highest, Average and Lowest fitness compared for 10000 generations. Some of them were able to finish in early generations.

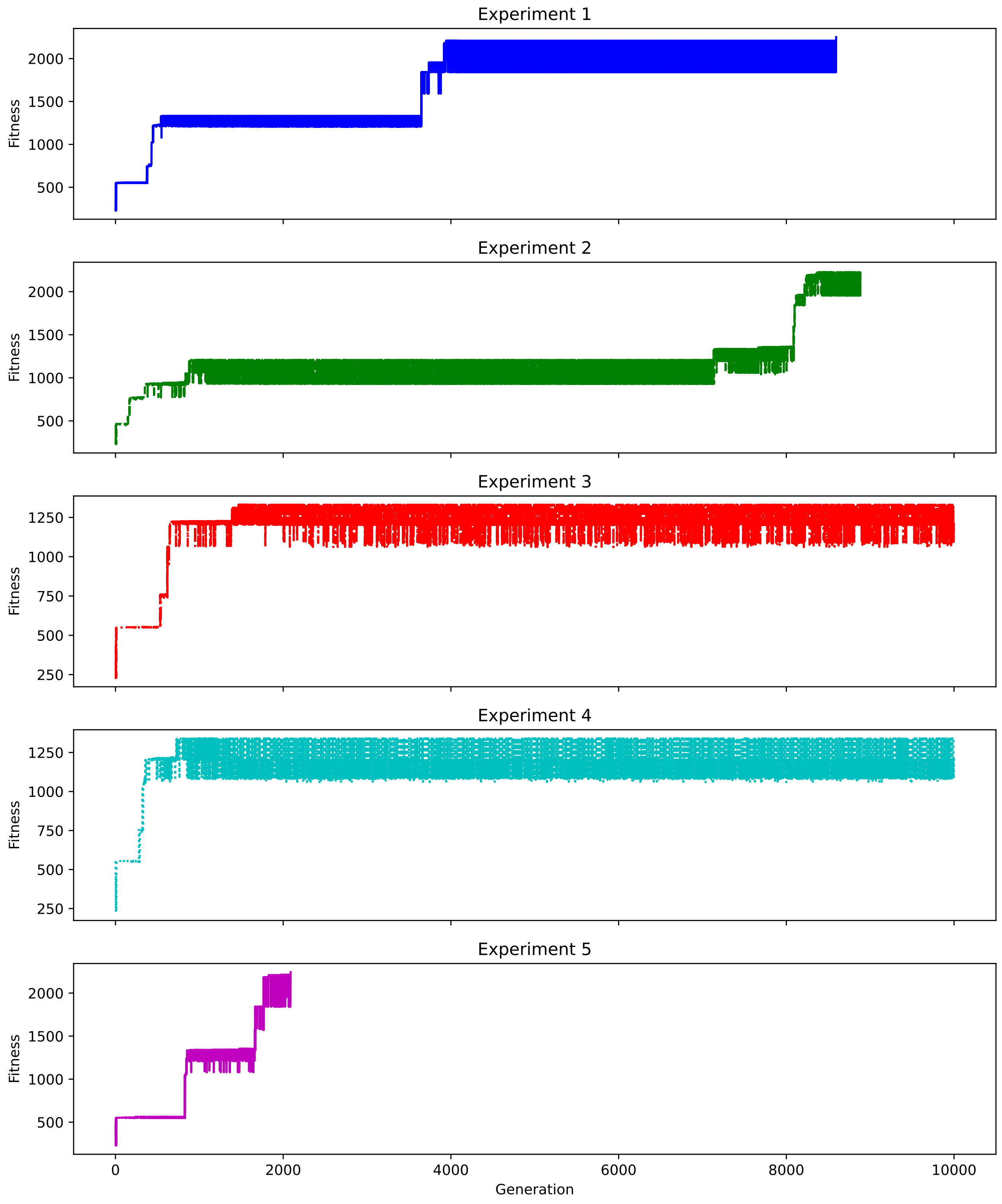


Figure 19: This plot shows five different experiments analysed separately. Overall some experiments were able to finish the level 3 gameplay.