Evolutionary Algorithm

William John Lautama

Model

Genotypes

Relevant Features

The primary feature that is considered is the sequence of actions that Mario can take. The only allowed actions that Mario can take are RightRun and RighRunJump. These action sequences are crucial since they directly influence Mario's behavior and performance in the game.

Encoding of Features

The way the action sequences are encoded is by putting it in an array of strings, with each string representing a specific action that Mario can perform. For example, "RightRunJump" represents Mario running to the right while jumping. This encoding allows for easy manipulation and comparison of different action sequences. Each population is also represented by a class called MarioDNA which have :

- A list of action strings (marioGenes)
- A MarioForwardModel class that captures the state of the game after executing the action sequence.
- A fitness score that defines the effectiveness of the action sequence.

Phenotypes

Instantiation of Genotype

The genotype (action sequence) is instantiated as a 'MarioDNA' object, which includes the action sequence and a clone of the game model. This instantiation allows the genotype to be evaluated in the game environment.

Parameter Justification

- Sequence Length (20) : This length balances the need for a diverse set of actions with computational efficiency. Ideally, this length can be more but due to the limitation of computation, it is limited to 20.
- Action Choices : Limited to 'RightRun' and 'RightRunJump' for simplicity and to focus on basic yet effective actions. This decision also allows for efficient computational workload. It is also worth mentioning that all levels that are presented for this assignment are able to be solved with just 2 of these inputs but the only problem is that it highly depends on the amount of population, generations, and the length of the sequence of actions.

Here is the pseudocode that initializes the genotypes based on the parameter provided :

```
ArrayList initialMarioGenes;
For (sequenceLength) {
     Add random allowed action to initialMarioGenes; return
initialMarioGenes;
```

This code is used at the start of the agent where the agent will define a random set of actions in the sequence of actions (RightRun or RightRunJump) and put it in the array. This then will be initialized in the initialMarioGenes variable.

Reproduction

There are 2 main reproduction methods that I used which are Crossover and Mutation.

Crossover

Crossover combines genes from two parents to create offspring. Two crossover points are chosen randomly, and genes are exchanged between these points. Here is the pseudo code of the crossover method :

```
ArrayList childGenes;
Int point1 = getRandomPoint;
Int point2 = getRandomPoint;

Int start = get lowest point;
Int end = get highest point;

For (i = 0; i < start) {
     childGenes.add(get parent1's genes)
}

For (i = start; i < end) {
     childGenes.add(get parent2's genes)
}

For (int i = end; i < parent's gene size) {
     childGenes.add(get parent1's genes)
}

Return MarioDNA(marioforwardmodel, childGenes);
```

Basically, it takes the parent1 and parent2 parameters. Then, a random point is defined from the variables. After that, it gets the highest and the lowest point for the bound of the crossover. Then, it adds the genes based on the crossover that was defined.

Mutation

Mutation randomly alters the genes in the offspring with a small probability, introducing new variations into the population. This is done to avoid local optima for the evolutionary algorithm so that it can still get the best solution even if it is stuck in one specific spot or problem. Here is the pseudocode for the mutation method :

```
If (random num < mutationRate) {
      For (gene size) {
            Set each gene to a random action (RightRun / RightRunJump)
      }
}
```

The genes are mutated if the random number that is defined is lower than the mutation rate. If it is lower than the mutation rate, it mutates all of the genes of that specific individual in the population.

Parent Selection

Selection Criteria

The selection criteria for both parents are completely random. This is to add more varied results for the children instead of just using the top result. Since the action sequences are only limited to 2 action sequences, this method is highly viable. From the random parents, it can make a variety of childrens making it easier for Mario to find the optimal child out of all the variance of data. Here is the pseudocode of the parent selection :

```
For (maxPopulation after elitism) {
     parent 1 = getRandom; Parent 2
     = getRandom;
```

This code may seem unfinished but it is only the first part of the code. The next part will be the offspring and the mutation part of the code which will be discussed in the next header.

Reproduction Frequency

Each selected pair of parents reproduces to generate one offspring. This keeps repeating until it replenishes the max population. This is because the parents are quite randomized and 1 offspring is enough for a variance of data for each pair. Since the parents are randomized each

time, the data that will be outputted will also be more randomized rather than using the same parent for more than one offspring. Here is the pseudocode for the offspring section (cont. Of Selection Criteria) :

```
      offspring = crossover;
      mutate(offspring);
      newPopulation.add(offspring);
}
```

Replacement Rule

Selection for Next Generation

The individuals that are selected for the next generation will be the best performing individuals. Since Elitism is incorporated, it depends on the total amount of elitism, for example, if elitismCount is 3, that means 3 of the best individuals will be part of the next generation. The rest of the generation will be from the offspring that was made from the parents through crossover and mutations. Here is the pseudocode to incorporate elitism :

```
For (elitismCount) {
      newPopulation.add(population.get(i));
}

The for loop for reproduction i starts from elitismCount;
```

Impact on Diversity

The impact of this method on diversity is the exploitation, exploration, and population diversity. The exploitation means that by retaining the top-performing individuals, the algorithm ensures that the best solutions are preserved and refined, promoting the exploitation of known good solutions. Exploration meaning that crossover and mutation introduces new genetic combinations and variations, promoting exploration of the fitness landscape to discover potentially better solutions. The last thing, population diversity meaning that elitism can help maintain high-quality solutions, but excessive elitism can reduce diversity. The crossover and mutation steps help mitigate this by introducing variability, ensuring that the population doesn’t converge too quickly on suboptimal solutions and continues to explore a wider range of potential solutions. The randomness of the parent selection also impacts the diversity of the results.

Fitness Function

Evaluation Metrics

There are a couple of evaluation metrics to define the fitness function. Here are all of the evaluation metrics :

- Mario's Vertical Position (Y- Coordinate) : This helps to determine how high Mario is. The higher mario is, the better fitness mario will achieve due to the nature of the levels.
- Completion Percentage : This helps determine the progress through the level. - Remaining Time : This encourages faster completion of the level.
- Game Status : Rewards the player on winning and penalizes losing or running out of time.
- Mario's Surroundings on the Y-Coordinate

Fitness Calculation

Here is the pseudocode for the fitness calculation :

```
Double fitnessNum;
Float marioY = getMario's Y position;

If (marioY < 1) {
      fitnessNum += 10000;
}

If (marioY > 1) { fitnessNum -=
      marioY * 100;
}

If (marioY > 140) { fitnessNum
      -= 100000000;
}

For (Below mario) {
      If (below mario is a brick) {
            fitnessNum += 10000;
            bDownBrick = true;
            break;
      }
}
```

```
If (bDownBrick) {
      bDownBrick = false;
      For (above mario) {
            If (above mario is a brick) {
                  fitnessNum -= 20000;
                  Break;
            }
      }
}

fitnessNum += CompletionPercentage * 100000;
fitnessNum += getRemainingTime;

If (win) {
      fitnessNum += 9999999999;
} else if (lose or timeout) {
      fitnessNum -= 9999999999;
}

Return fitnessNum
```

The fitness function starts by putting mario Y's position into a variable. Then, I noticed some levels have a specific spot in the top that Mario can just walk on so that Mario can skip over the level. Therefore, I added an implementation that rewards mario if mario is above the screen making mario always jump on the top whenever mario's starting position is at the top. It also checks if mario is below the bottom platform and if mario is below the bottom platform, its the same as calling mario dead. The next part is checking if there is a brick below mario. If there is a brick, mario will be rewarded since mario will land on the brick if mario is in the air. If not, then Mario will not get any reward.

The next part is the 2nd part of the check. If there is a brick below mario but there is a brick above mario, that means mario could've jumped on the brick above him which means mario will be penalized. Then, the completion percentage is also checked with a very high number so that Mario prioritizes completing the level. The remaining time is also checked to ensure that Mario finishes the level as fast as possible. The last check that the fitness function goes through is whether Mario wins or loses. If mario wins, mario gets rewarded heavily since that is the main objective of the game but if mario loses, mario gets penalized heavily since I don't want Mario to lose the level.

Training Approach

Initial Population

Generation Method

The initial population is generated randomly. Each individual in the population (an instance of ‘MarioDNA’) is initialized with a random sequence of actions (‘marioGenes’). This sequence is of a fixed length, defined by the parameter ‘sequenceLength.’

Number of Organisms

The number of organisms in the initial population is determined by the parameter ‘maxPopulation’, which is set to 30 in this implementation.

Here is the pseudocode of the initialize function :

```
For (maxPopulation) {
     MarioDNA marioDna = new MarioDNA(clonedModel,
initialiseMarioGenes()); population.add(marioDna);
}
```

Basically, it takes in the variable maxPopulation and then runs a for loop based on how many max population it is. Then, it makes an individual using the MarioDNA class and adds it into the population.

Parameter Selection

Population Size

Value : maxPopulation = 30

A population size of 30 is a balanced choice that allows for sufficient genetic diversity while keeping the computational load manageable. This size helps to explore the solution space adequately without overwhelming the system with too many simulations. This might seem small but due to the nature of the action sequence method, it is crucial for it to not be too much. Ideally, more population would be beneficial but if the game runs too slow, it could affect the performance of the agent as well. The impact of having more population size wouldn’t be worth the performance compromise.

Sequence Length

Value : sequenceLength = 20

A sequence length of 20 provides a good balance between having enough actions to complete meaningful tasks and maintaining the ability to evaluate and evolve individuals efficiently. This length ensures that the agent can perform a variety of actions while keeping the complexity of evaluation reasonable. Ideally, more sequence length would be beneficial but if the game runs too slow, it could affect the performance of the agent as well. The impact of having more sequence length might be worth the performance compromise but to a certain extent and after trial and error, the number 20 is a good base number for the sequence length.

Number of Generations

Value : ‘generations = 30’
Training over 30 generations provides enough iterations for the evolutionary algorithm to improve the population significantly. This number is chosen to balance between achieving convergence and maintaining reasonable training time. The impact of having more generations wouldn’t be worth the performance compromise and 30 is already a good number to base upon.

Mutation Rate

Value : ‘mutationRate = 0.05’
A mutation rate of 5% is low enough to maintain the integrity of high-performing individuals but high enough to introduce new genetic material into the population. This rate ensures that the algorithm can explore new solutions while refining existing ones. Since I want the algorithm to be predictable but the minority of the population would have a chance to escape potential local optima, 5% is a good rate of mutation to implement.

Elitism Count

Value : ‘elitismCount = 3’
Retaining the top 3 individuals in each generation ensures that the best solutions are preserved and passed on to the next generation. This helps in maintaining the quality of the population and speeds up convergence. 3 is also a good number since the total number of the population is 30 meaning that 10% of the new population are the individuals that was brought upon from the previous population through elitism.

Crossover Points

Value: Random crossover points within the sequence length.
Using random crossover points introduces variability in the offspring, promoting genetic diversity. This strategy helps in exploring the solution space more effectively.

Termination Criteria

The algorithm is set to run for a fixed number of generations, which is 30 in this case. This choice is made to ensure that the agent has sufficient time to evolve and improve while keeping the training process within practical time limits.By observing the performance of the best individual in each generation, one can assess whether the population is converging towards an optimal solution.

If the performance stabilizes or shows diminishing returns, it indicates that the population has potentially reached a local or global optimum. Running for a fixed number of generations ensures that the training process does not run indefinitely, making it feasible to complete within available computational resources and time constraints. If the algorithm reaches local or global optimum, the agent will still be the best performing fitness individual but it will meet its demise / will not progress.

Evaluation

Datasets

The evolutionary algorithm was trained using 16 different levels. These levels are 1-1, 1-2, 1-3, 2-1, 3-1, 3-3, 4-1, 4-2, 5-1, 5-3, 6-1, 6-2, 6-3, 7-1, and 8-1. These levels have a variety of challenges of its own ranging from levels with a lot of grounded platforms to levels that have huge gaps in between the platforms. There are also 2 levels which are 1-2 and 4-2 that have a straight platform at the top that Mario can just run on to skip the whole level. The way the algorithm works is that it generates the training while Mario is doing the levels since it doesn't do any offline training. The disadvantage of this is that Mario cannot have perfect genes at the start but this is counteracted by tha advantage which is the adaptability and also the smaller time consumption of running the algorithm.

Performance

The effectiveness of the algorithm was evaluated based on the fitness function, which considered various factors such as Mario's position, completion percentage, remaining time, and game status (win or loss).

Exploration of the Fitness Landscape

- Initialization: The initial population was generated with random sequences of actions (RightRun / RightRunJump).
- Selection: The fittest individuals from each generation were selected using a fitness function.

- Crossover and Mutation: New generations were created through crossover and mutation, maintaining genetic diversity and allowing the exploration of new parts of the fitness landscape.

Control for Overfitting / Underfitting

- Elitism: Carrying over the top-performing individuals to the next generation ensures that high-quality solutions are retained.
- Mutation Rate: A mutation rate of 5% was chosen to balance exploration and exploitation. Too high a rate might lead to random search, while too low a rate might cause premature convergence.
- Generations: Evolving over 30 generations provided a balance between computational feasibility and solution quality.

Evaluation & Reflection

Experimental Process

There are a couple of things that I experimented during the process which are :

- Probability Gene Implementation
  Before going with the sequence of actions implementation, I did an implementation where Mario will have an array of probability for his actions and the probability evolves overtime as Mario is completing the levels.

- Adding Enemy Kills as Fitness Function
  I initially added enemy kills as one of the fitness functions but didn't use it at the end because enemy kills are not part of the main objective but it still affected the gameplay in some way in a negative way.

- Using All The Available Actions
  Originally, I used all the available actions but I didn't end up using it because it affected the time and the completion greatly.

- No Upper & Lower Mario-Y Checks
  I didn't implement an upper and lower check for Mario for a brick before but when I tried implementing it and it turned out to be better, I decided to implement it.

- No Height Fitness Function
  Originally, I didn't have a fitness function that rewards mario the higher he is but after I implemented it, mario is way more efficient therefore I included it.

These approaches' results will be put in the next section for each level.

Results

The results of this experiment is from running the level 2 times and getting the best results. If the level is not completed, the time left will be marked as N/A.

Experiment Type	Level	Completion Percentage	Status	Time Left
Probability Gene Implementation	1-1	35.19%	Lose	N/A
	1-2	35.75%	Lose	N/A
	1-3	12.96%	Lose	N/A
	2-1	43.92%	Lose	N/A
	3-1	20.63%	Lose	N/A
	3-3	9.24%	Lose	N/A
	4-1	49.17%	Lose	N/A
	4-2	8.77%	Lose	N/A
	5-1	19.66%	Lose	N/A
	5-3	9.14%	Lose	N/A
	6-1	92.23%	Lose	N/A
	6-2	18.68%	Lose	N/A
	6-3	17.86%	Lose	N/A
	7-1	39.27%	Lose	N/A
	8-1	39.27%	Lose	N/A
Adding Enemy	1-1	100%	Win	50

Kills as Fitness Function	1-2	100%	Win	53
	1-3	100%	Win	53
	2-1	93%	Timeout	N/A
	3-1	100%	Win	51
	3-3	92.89%	Lose	N/A
	4-1	100%	Win	48
	4-2	100%	Win	52
	5-1	100%	Win	51
	5-3	100%	Win	52
	6-1	100%	Win	51
	6-2	100%	Win	49
	6-3	100%	Win	53
	7-1	100%	Win	49
	8-1	100%	Win	40
Using All the Available Actions	1-1	71.34%	Timeout	N/A
	1-2	100%	Win	52
	1-3	84.63%	Timeout	N/A
	2-1	33.39%	Timeout	N/A
	3-1	16.74%	Timeout	N/A

	3-3	34.32%	Timeout	N/A
	4-1	8.46%	Timeout	N/A
	4-2	100%	Win	50

	5-1	20.63%	Timeout	N/A
	5-3	36.13%	Timeout	N/A
	6-1	20.12%	Timeout	N/A
	6-2	19.16%	Timeout	N/A
	6-3	18.80%	Timeout	N/A
	7-1	86.94%	Timeout	n/A
	8-1	20.60%	Timeout	N/A
No Upper & Lower Mario-Y Checks	1-1	100%	Win	49
	1-2	100%	Win	53
	1-3	100%	Win	53
	2-1	93.21%	Timeout	N/A
	3-1	100%	Win	52
	3-3	92.89%	Lose	N/A
	4-1	100%	Win	49
	4-2	100%	Win	52
	5-1	100%	Win	51

	5-3	100%	Win	53
	6-1	100%	Win	51
	6-2	100%	Win	49
	6-3	100%	Win	53
	7-1	100%	Win	51
	8-1	100%	Win	42
No Height Fitness Function	1-1	100%	Win	50
	1-2	100%	Win	53
	1-3	48.19%	Lose	N/A
	2-1	93.21%	Timeout	N/A
	3-1	100%	Win	51

	3-3	40.62%	Lose	N/A
	4-1	100%	Win	47
	4-2	100%	Win	52
	5-1	100%	Win	51
	5-3	100%	Win	53
	6-1	100%	Win	50
	6-2	100%	Win	48
	6-3	100%	Win	52

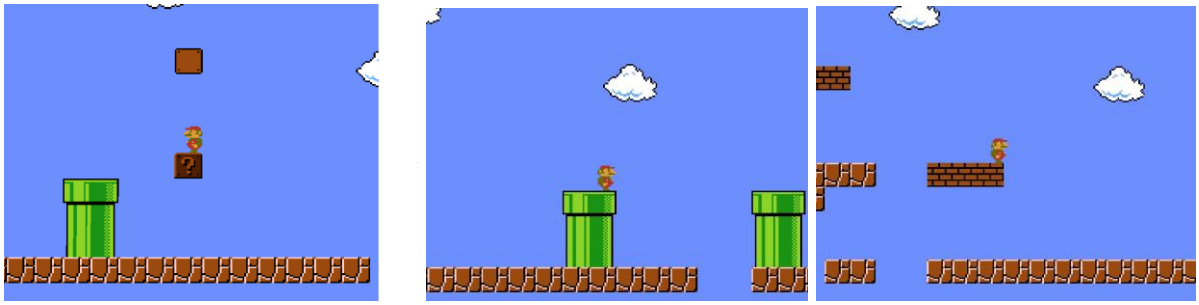
	7-1	100%	Win	51
	8-1	100%	Win	41
Finalized Version	1-1	100%	Win	50
	1-2	100%	Win	53
	1-3	100%	Win	53
	2-1	93.21%	Timeout	N/A
	3-1	100%	Win	51
	3-3	92.89%	Lose	N/A
	4-1	100%	Win	48
	4-2	100%	Win	52
	5-1	100%	Win	50
	5-3	100%	Win	53
	6-1	100%	Win	51
	6-2	100%	Win	50
	6-3	100%	Win	53
	7-1	100%	Win	51
	8-1	100%	Win	42

Reflection

From each experimentation, it made a feasible solution to make the AI better. The gene implementation works well enough for Mario to work but not well enough for Mario to finish the levels at all. Adding enemy kills as a fitness function is not really beneficial as well since it slows

down the completion time of each level. If Mario is using all of the available actions, Mario gets timed out very easily.

This is due to the fact that when Mario is standing on a tall object and the next part of the world is not as tall / taller than the object, Mario won't move forward and will stay still with buttons that don't really move Mario.



When the upper & lower checks for Mario are not implemented, the result of it is not far from the finalized version. However, the checks are meant to act as a failsafe just in case Mario could've done better. Adding the height check in the fitness function really affects the agent on specific levels, especially levels with a lot of gaps. Since Mario would be more advantageous if he is upwards, it makes sense why Mario would perform better if the higher Mario is, the more reward Mario gets.

The finalized version is arguably not the best solution since it cannot finish level 2-1 at all due to the wall, and cannot finish 3-3 consistently, it is still a viable solution due to the speed at which Mario finishes the levels.

Future Work

There are a couple of things that can be done in future work to give improvements to the algorithm for further benefits which are :

- Action Set Expansion: Incorporating more complex actions (e.g. combinations involving jumps and runs with timing considerations) could improve the agent's performance on more challenging levels. This could be incorporated by adjusting the fitness function overall as well.
- Dynamic Fitness Function: Adapting the fitness function to give more weight to strategic behaviors depending on how the level is shaped. Since some level doesn't need specific fitness functions and is better off without the fitness function, it would be beneficial if it can be adjusted dynamically.
- Hybrid Approaches: Combining evolutionary strategies with other strategies instead of just action sequences such as evolving body parts, neural networks, and more could be beneficial and could improve the agent further.

Conclusion

The evolutionary algorithm for training a Mario-playing agent, developed with the constraints of using only two actions (“RightRun” and “RightRunJump”), demonstrates a promising approach to solving levels in a computationally efficient manner. The process of defining genotypes as sequences of actions and evolving them through crossover and mutation has proven effective, particularly with the incorporation of specific fitness function metrics such as vertical position, completion percentage, remaining time, and game status.

References

Baldominos, A., Saez, Y., Recio, G., & Calle, J. (2015). Learning Levels of Mario AI Using Genetic Algorithms. *Advances in Artificial Intelligence*, 267–277. https://doi.org/10.1007/978-3-319-24598-0_24

Evolving a Neural Network to play Super Mario Bros using Neuroevolution of Augmenting Topologies. (n.d.). Www.googlesciencefair.com. Retrieved June 2, 2024, from https://vivek.lol/super-mario-neat/

Togelius, J., Karakovskiy, S., & Baumgarten, R. (n.d.). *The 2009 Mario AI Competition*. Retrieved June 2, 2024, from http://julian.togelius.com/Togelius2010The.pdf

Togelius, J., Shaker, N., Karakovskiy, S., & Yannakakis, G. N. (2013). The Mario AI Championship 2009-2012. *AI Magazine*, *34*(3), 89. https://doi.org/10.1609/aimag.v34i3.2492

Shaker, N., Yannakakis, G. N., Togelius, J., Nicolau, M., & O’neill, M. (2012). Evolving personalized content for super mario bros using grammatical evolution: 8th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2012. *Proceedings of the 8th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2012*, 75–80. https://nyuscholars.nyu.edu/en/publications/evolving-personalized-content-for-super-mario-brosusing-grammati

Volz, V., Schrum, J., Liu, J., Lucas, S. M., Smith, A., & Risi, S. (2018). Evolving Mario Levels in the Latent Space of a Deep Convolutional Generative Adversarial Network. *ArXiv:1805.00728 [Cs]*. https://arxiv.org/abs/1805.00728