Playing Super Mario World With A Neural Network Created From Various Genetic Algorithms

Joseph Spielman068@umn.edu

December 16, 2018

Abstract

This project develops neural networks that play the game Super Mario World on emulation software. The neural networks are generated through various genetic algorithms, Original(GA), Breeder(BGA), Elitism(EGA) and Adaptive(AGA). The neural networks and genetic algorithms are components of a NeuroEvolution of Augmenting Topologies algorithm. 5 trials were conducted on the level Donut Plains, each variation ran until the 50th generation. Of the variations only the Breeder Genetic Algorithm successfully completed the level. Due to the limited amount trials the overall effectiveness of each variation is not fully understood. Each variation contained varying degrees of success as agents completed more than 50% of the level.

Introduction

Super Mario World is 2D platformer released in 1990 for the Super Nintendo Entertainment System[12]. The player takes control of the character Mario and is tasked with reaching the end of every level. The player initially starts at the left most position of the level, with the end of the level being the point farthest to the right. To get to the goal, the player most hop on platforms, jump over pits, and evade enemy attacks. There are coins and power ups which can be collected. Gather enough coins and the player gains an extra try at a level, and the power ups allow the player to take damage from an enemy. The player is able to move left, move right, jump and if they have one, use their power up.

This project will try to create a neural network which is able to successfully complete a single level of Super Mario World. The neural network will be generated by a NeuroEvolution of Augmenting Topologies or NEAT algorithm. If this project is able to successfully create an agent capable of completing a level, this may be a solution to an issue of computer controlled teammates making poor choices, known as static behavior[15]. These poor decisions may cause the teammate to not meet the players expectations, hindering the overall experience[15]. Specifically static behavior is when a teammate only looks at part of the information inside of the environment[15]. An example of this would be if out agent walked to the right to reach the goal, but did not jump over an enemy. Here the agent only took into account the goal position and not the enemies. If the agent is able to complete the level, it must have taken in sufficient information from the environment to make proper choices. With testing and creating new networks the agent should take into account all important elements of the game world.

Another use for this solution is to create a new experience for players, where their computer controlled teammates learn the levels with them. While this would have not been possible with the hardware when Super Mario World was released, current gaming consoles does possess the necessary processing power[4][5].

Creating an agent that is able to move to the right and jump over non-moving obstacles could be accomplished through a simple reactive agent, which always moves to the right checks for gaps between platforms, jumping when necessary. The levels of Super Mario World are significantly more complex, containing moving platforms, moving enemies, plus the additional incentive to gather coins and resources. This will require the agent to take into account multiple elements of the environment simultaneously. The agent must also be able to handle a changing environment, such as when an enemy or platform changes position.

Related Work

Deviations of Genetic Algorithms

Because a NEAT algorithm is used to solve this project's problem. A large component of how successful the NEAT algorithm is come from its implemented genetic algorithm. Because of this, it is important to look at genetic algorithm variations and how these variations altered the algorithms and the corresponding performance changes.

Global Parallel Genetic Algorithm (GPGA)

GPGA unlike the other two variations does not alter the fundamental process of GA. Instead it alters GAs structure to take advantage of computer architecture[8]. One version of GPGA is to have the individuals be placed in shared memory and run evaluations from several processors at the same time to speed up returning results[8]. Another approach is to have a master processor and make the remaining processors be slaves[8]. The master processor then sends individuals off to be evaluated to one of the slave processors. Both versions are a divide and conquer approach to the genetic algorithm, rather than having all of the process run on a single core. There are some issues with this implementation.

While GA has a straightforward implementation, new issues can arise from parallel computing. One being that you must make sure generations are synchronized, even if one processor is finished, it must wait for others to finish. Implementations of this approach found issues with overhead cost gradually increasing, this overtime reduced the benefit of the initial speed. The overhead is due to the sending of information between processors. The implementations the information was gathered from were conducted on hardware from the 1990s. The overhead issue may no longer be present in current hardware, or not as prevalent, however this is only speculation.

Breeder Genetic Algorithm (BGA)

GA is based on the idea of natural selection, while BGA is based on the concept of artificial selection, where a breeder selects who is allowed to mate[13]. Like artificial selection, BGA introduces the concept of a breeder to the algorithm. The breeder decides who should and should not mate. To represent the breeder, a new variable is added, truncate. This variable is the percentage of top individuals to be used to populate the next generation. Rather than having all individuals above a

threshold be able to mate, only the fittest of the generation are allowed to populate the next. The truncated population genes is used to create all of the offspring for the next generation. With the addition of the truncate variable, a new issue can arise. If you set the truncate to a low percentage, is highly likely the population will converge quickly. This convergence can lead

to a plateau, stopping the algorithm from finding a better solution. Increasing the population can reduce the time before a population converges on itself, due to the truncated population containing more individuals. Which can help offset a low truncate percentage. Because BGA only takes the top individuals diversity can be lost. This loss of diversity makes BGA more susceptible to get stuck at a local maximum.

Genetic Algorithm with Varying Population Size (GAVaPS)

When using GA the population size must be set. This variable controls the size of each generation. When setting this variable a trade off between algorithm run-time and answer correctness must be considered. GAVaPS removes the population parameter and takes a dynamic approach to population size[6]. Rather than always having a generation of a set population size, GAVaPS changes the population size of each generation. GAVaPS removes the fitness concept and evaluates each individual once. At evaluation the individual is given a lifetime, representing the number of generations it is allowed to exist. There is no fitness selection, all genes can populate with one another. This method requires that a function is provided to determine the lifetime of an individual, this is likely to be similar to the fitness evaluation function. Unlike GA which assigns a fitness to each individual and checks every generation for who is allowed to mate. GAVaPS finds a solution because individuals with longer lifetimes are able to create more offspring which passes on their genes, while those with lesser lifetimes are unable to create as many offspring. GAVaPS introduces a new variable, reproduction ratio. Reproduction ratio is multiplied by the current population to find the number of offspring to be created. This introduces a new issue of finding the proper reproduction ratio, the study suggests that 0.4 is sufficient enough to gain results better than standard GA[6]. Using the lifetime functions from the study, a trend was that at the start, there was a large population increase and then as time went on the population slowly decreased. The large population at the start can create an issue if you do not possess the proper amount of memory to store all of the information. GAVaPS with lifetime allocation strategy provided better performance than GA in test-cases. Though it provided better results, extra work must be performed for the lifetime function in order for GAVaPS to function properly and be more efficient.

Adaptive Genetic Algorithm (AGA)

AGA removes the mutation and cross over parameter[14]. Instead these parameters are calculated throughout the algorithms execution. Before a child offspring is created the mutation rate and cross over rate is calculated. When the offspring's parents are of poorer results the cross over and mutation rates are set higher. This insures that good answers are not altered heavily and poorer ones are changed to possibly provide better solutions. Depending on the problem you may need to alter the equations within this algorithm. If your problem has many local maximum, a higher mutation may be needed to find the actual maximum. Due to the needing to modify the equations, the overall implementation of this algorithm possibly more complex than GA. Because mutation and cross over rates are not static, you may be able to achieve a more appropriate mutation and cross over rate for each offspring which could not be achieved with GA.

Elitism Genetic Algorithm (EGA)

EGA is a simple modification, rather than having the next population only consist of offspring from the previous generation, this variation takes a top percentage of the population and puts them in the next generation[7]. This alteration ensures that the quality of the answer does not decrease from one generation to the next. Due to EGA allowing the top percentage of a population to breed and then be part of the next generation, premature convergence is more common. Premature convergence is able to be reduced through the combination of EGA and GPGA. Though EGA stops the solution quality from going down, another solution to this issue is by storing the solution with the maximum fitness at all times and changed the stored solution if another member of a population is of higher quality. The alternate solution does require additional memory, for the extra stored solution.

Other Approaches

A* Approaches

Moving a character in a game world from start to finish could be achieved through the A* algorithm. A* is a greedy shortest path algorithm. A* looks at path values as well as a value called a heuristic. A* combines a path's value with its heuristics value, from this it will choose the path with the smallest sum[10]. A* is able to have multiple heuristics values. For our problem heuristics could be direct distance to goal, or how many enemies are on a particular path. Heuristics will need to be updated regularly as some enemies are mobile and may move in or out of a path. Special modifications must be added to A* to handle this problem's environment. The environment is dynamic, meaning A* must be able to update the heuristics when an enemy or platform changes location. One variation of A* which achieves this is Dynamic Repairing A*[9]. DRA* repairs path options when information has changed, this would handle moving elements within the game world. The environment is not set up as a graph or tree, which is required for A* or DRA*.Multiple paths for A* to choose from would need to be generated.

Q-Learning Approaches

Q-Learning has been successfully applied to other game environments such as the Atari[11]. Q-Learning handles the environment with states, actions and rewards[16]. An agent is rewarded for its action, an action's reward can decrease or increase depending on the state it was performed in. The goal of Q-Learning is to create a policy function which solves the given problem. Though actions have reward value, the agent is unaware of them initially and must learn through trial and error. This information is stored inside of a Q table, which contains values for actions depending on the state. Because the Q table is empty initially, the agent does exploration to try and piece together information about the environment. Once the Q table has been partially filled out the agent can begin making informed decisions in order to solve the problem. Unlike NEAT, Q- Learning creates an agent that chooses actions which lead to a goal, while NEAT implementations create an agent that reacts in a way that leads to a goal. Implementing this problem would require that actions inside of game world have corresponding value. Depending on the amount of states and actions within the game world a large amount of evaluation functions may need to be created. This method is likely to create a versatile agent than NEAT as in being able to handle multiple levels however, NEAT is a simpler algorithm to implement.

Approach

This project will solve a level in Super Mario World through a NeuroEvolution of Augmenting Topologies algorithm. The overall approach of a NEAT algorithm combines neural networks with genetic algorithms. The neural network is generated by the genetic algorithms, because of this, the NEAT is highly dependent on the ability of the genetic algorithm to generate successful solutions. The neural network is represented by genes within the population, this includes both nodes and connections. Several genetic algorithms will be implemented in order to find the most efficient variation.

Neural Networks

A neural network is a graph structure, containing nodes typically arranged in layers. Given information, a node will "fire" depending if the information given is above a threshold or it will "not fire" if it is below it. Nodes can have multiple inputs and each input is given a weight. **Connections** represent where a node is acquiring input from. **Weights** determine how much a given input influences the neuron to fire. The lower the weight the lower the influence. The output of the node can either be sent to another node in a deeper layer, or it can be sent as output to have the agent perform an action. The neural networks for this problem decides if the agent presses a button on the controller, each button corresponds to an action the character can perform.

Genetic Algorithms

Fundamentally a genetic algorithm emulates biological evolution[17]. A genetic algorithm begins by creating an initial population, each member being randomly generated neural networks. The generated neural networks represent the genes of each member. These neural networks are tested and assigned a fitness value based on their quality of solution, the higher the fitness, the higher the quality. Then members of the population are used to generate the next generation, those with higher fitness are allowed to generate more offspring. The same process of testing and creating the next generation is done on the offspring, this is continued until a desired fitness is achieved or the generation limit is exceeded. The NEAT algorithm retains the individual with the highest fitness at all points, even if it is not present within the current population.

Parameters

When an offspring is created mutation and crossover occur. Crossover is the combination of both parents genes, the higher the crossover rate the higher chance the offspring receives genes from the second parent . Mutation is when the genes of the offspring are randomly changed, being completely distinct from both parents. There are several different mutations that can occur on the neural network.

Crossover Rate chance of giving the offspring genes of the second parent.
Connections Rate is the chance that a weight from a given input for a node is altered.
Link Rate is the chance of new input being added to a node.
Bias Rate is the chance of altering an existing input to a node.
Node Rate is how likely a new node will be generated.

GA Variations

Four genetic algorithm variations BGA, EGA, AGA and the original GA will be implemented.
Primary difference from the original GA:
BGA only allows the top percentage of individuals to create offspring rather than allowing all members be given a chance to create offspring.
Top Percentage is the percentage of top members that are allowed to breed.

EGA takes the top members of the last generation and puts them in the next generation. EGA has additional parameter.
Elite Population numerical amount of top members to be brought over to the next generation.

AGA does not require input for any of the mutation rates or the crossover rate. Instead these parameters are calculated for each member.
Mutation rate if a parent is above the average fitness:

```
calculation = maxFitness - larger_parent.fitness
calculation = calculation / (maxFitness - averageFitness)
calculation = calculation * 0.5
MUTATION_RATE = calculation
```

Mutation rate if both parents are below the average fitness:
0.5

Crossover rate if a parent is above the average fitness:

```
calculation = maxFitness - larger_parent.fitness
calculation = calculation / (maxFitness - averageFitness)
CROSS_OVER_CHANCE = calculation
```

Crossover rate if both parents are below the average fitness:
1
Because the NEAT algorithm contains additional mutations rates and those mutation rates are not all the same value, the calculation values are adjusted.

```
Connection_Rate = MUTATION_RATE / 2
Link_Rate = MUTATION_RATE * 4
```

Experiment

Design Of Experiment

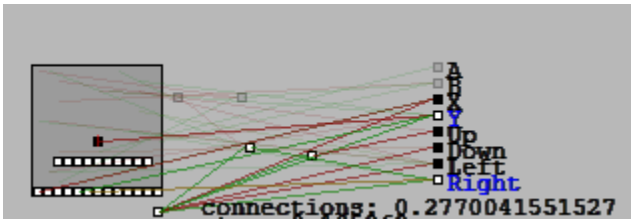
This experiment will be conducted using the Bizhawk Emulator[1]. This emulator will be used to simulate the hardware Super Mario World was designed to run on. The code running the NEAT algorithm will be a modified version of available code that creates Neural Networks to play Super Mario World[3]. The code is written in the Lua programming language and is executed with the Bizhawk Lua console[2]. The code will be modified to implement BGA, EGA, AGA and the original GA.

Because genetic algorithms do use randomness several trials of each algorithm will be ran, this is done in order to limit the chance of an algorithm selecting the best randomly and skewing results. Each variation will be ran 5 times until the 50th generation is complete. Below are the parameters used for this experiment:

```
Population = 100,
MutateConnectionsChance = 0.25,
CrossoverChance = 0.75,
LinkMutationChance = 2.0,
NodeMutationChance = 0.50,
BiasMutationChance = 0.40,
--For EGA and BGA Respectivly
ElitePopulation = 10
TopPercentage = 50
```

All of the trials will be performed on the same level Donut Plains. The fitness of each individual will be measured based on how far to the right of the level they travel, the amount of coins collected, the amount of power ups collected and the number of enemies defeated, each of these actions being worth 50 fitness points. Upon completion of the level an additional 1000 fitness points are given.

Example Neural Network



The neural network receives input through two types of connections. A enemy connection(red) and a platform connection(green). The enemy connections send information to a node if an enemy, represented by the black blocks, goes through that connections sensor. The platform connection provides information if a platform, represented by the white blocks, goes through it's sensor. The sensors are randomly placed around the screen when the connection is created. Sensors are not placed throughout the level, instead they are placed with in the character's view of the elements, moving with the character. The agent looks directly at the game world and has direct access to the world's information. The agents view is limited to the information that is able to be displayed on screen, as in what a typical player would see. The agent is able to signal a button press to the emulator. The emulator will be ran at 6400% normal speed in order to quicken data collection.

A Mario is played by each member of a generation, this Mario is played until either it loses a life, stands still for 2 seconds(game time) or completes the level. Each neural net plays the level once. After all of the members have played the level the next generation is created. This experiment will log the maximum fitness of each generation.

The primary goal of the experiment is to assess the various genetic algorithms and find the most effective solution in terms of generations used to finish the level.

Results

Individual Performances

Compared Averages

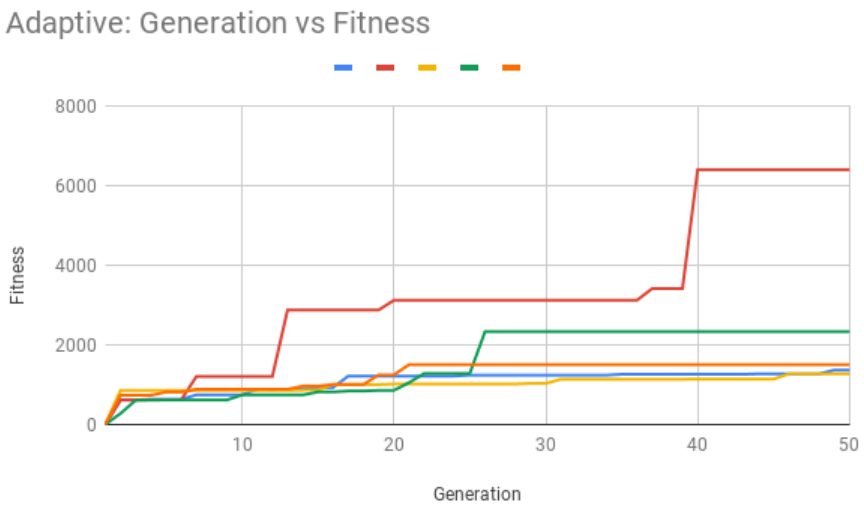


Figure 1: Contains the maximum fitness for each generation from 1 to 50 for the adaptive GA for 5 trials.

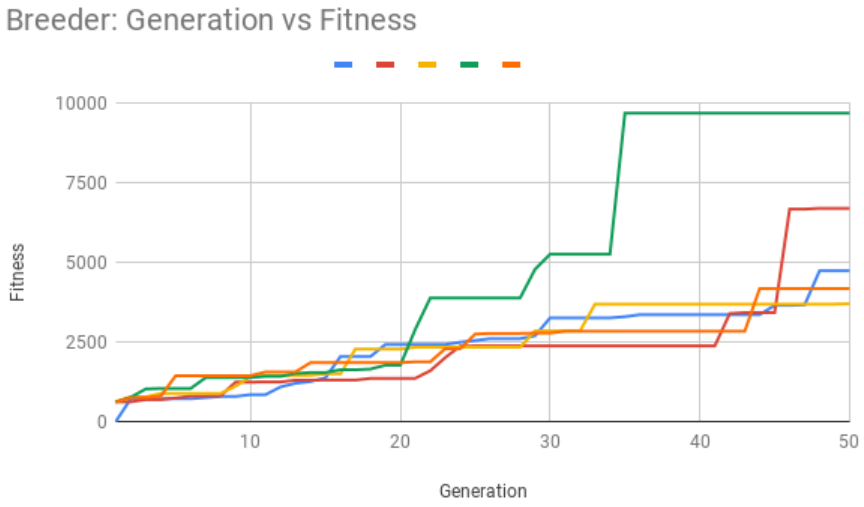


Figure 2: Contains the maximum fitness for each generation from 1 to 50 for the Breeder GA for 5 trials. The breeder algorithm produced a member capable of completing the level in the trial represented by the green line.

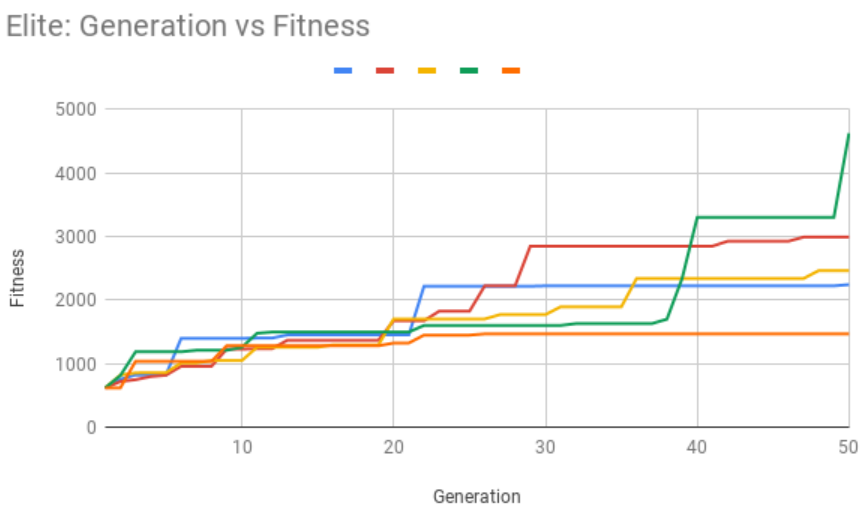


Figure 3: Contains the maximum fitness for each generation from 1 to 50 for the Elitism GA for 5 trials.

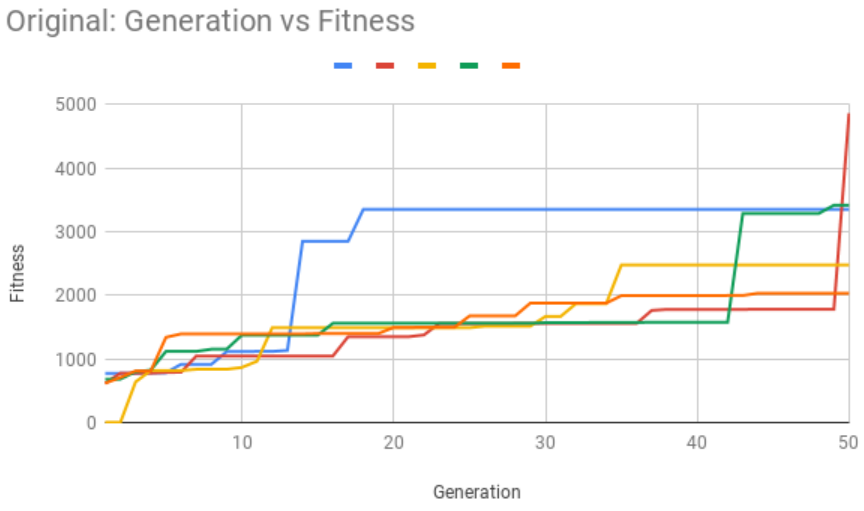


Figure 4: Contains the maximum fitness for each generation from 1 to 50 for the original GA for 5 trials.

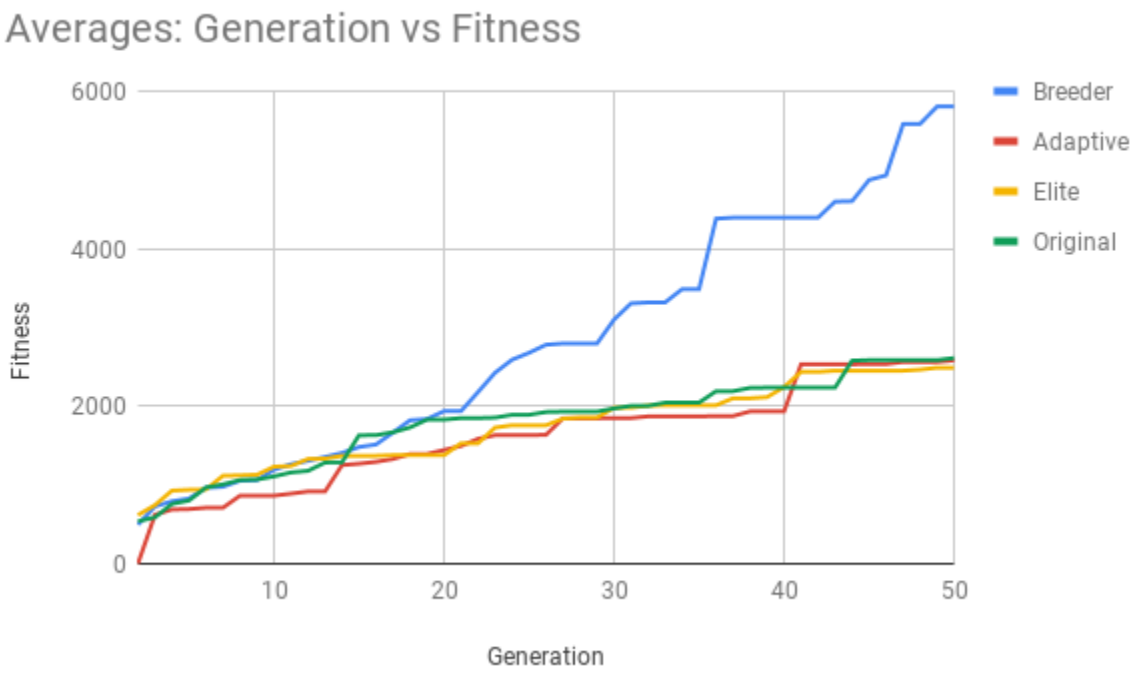


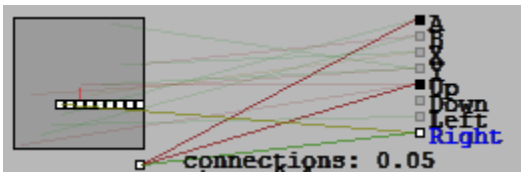
Figure 5: Contains the average maximum fitness for each algorithm variation.

Analysis

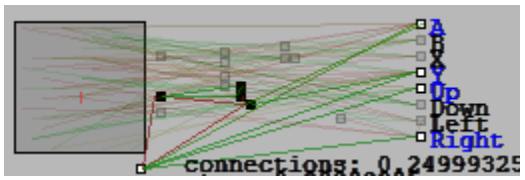
Adaptive:
The AGA was able to produce an offspring of higher quality than any of the trials for the GA and EGA, with a fitness greater than 6000(Figure 1). This only occurred once, while the remaining 4 trials only produced a fitness around 2000 a fitness similar to the average of GA and EGA(Figure 4)(Figure 3). Due to the small amount of trials, the successful trial of AGA could have occured because it generated a strong solution randomly, rather than it being generated due to AGA's unique properties. From the trials AGA generated simpler solutions compared to the other variations, having less connections and nodes for any given generation. This is the result of the equations used to generate the mutation and crossover rates. The better the solution the lower the mutation rate. Unlike problems with a set amount of genes, the mutation rate in the NEAT algorithm is tied to the creation of new neurons, this inevitably leads AGA to create simpler solutions.
Breeder:
The BGA algorithm was the only algorithm to successfully complete the Donut Plains level and had higher averages than the other variations(Figure 5). The higher averages are likely because it only allows strong members of a population to repopulate. This allows the passing of useful neural connections, such as having the Mario walk to the right, to be more numerous in the population than the other variations. This creates relatively similar populations, like the EGA, however it does not do direct copies, this allows for more mutations and unique solutions to be tested. The BGA averages have been partially skewed due to the structure of this experiment. In the trial BGA completed the level it was granted an additional 1000 points. These additional points increased the

score drastically, for doing relatively the same behavior, that is, moving to the right of the screen, where the position happened to be the end of the level. The limited amount of trials may have over inflated the average. Even taking this into account, the other four trials of BGA still had a higher average than the other variations.

Adaptive Neural Network, Generation 45



Breeder Neural Network, Generation 45



Elite:
The EGA has the overall poorest average, though it is not by a large amount(Figure 5). This poor performance is likely due to the copying of individuals from the previous generation to the next. Though it allows for successful answers to repopulate and stops the chance of the answer's quality from decreasing, it actually decreases the unique population for each generation. For this project 10 members were saved from the previous generation, meaning only 90 offspring were created for the next. By the end of each trial, 500 more offspring would be generated and tested in the other variations, reducing the chance of mutating a more successful solution. EGA's data contains multiple large plateaus(Figure 3), likely due to the reduction in solution variation. Another issue with the EGA is that the NEAT algorithm already handles one of the issues EGA was designed to fix. The NEAT algorithm, holds onto the solution with the maximum fitness, meaning the algorithm will never return a lower quality solution from the previous generation to the next. One of the benefits of EGA is that the highest quality solution is always returned.
Original:
GA has similar averages to EGA and AGA(Figure 5). This may be caused by how the next generation is populated. For these 3 algorithms all members of a population are able to breed for the next generation. Because poor solutions help create the next generation, the next generation may contain bad node connections which can completely nullify the good genes. Such as adding a gene that has the Mario stand still when an enemy is moving towards it. While it may contain portions of a good solution, this single bad connection may heavily reduce it's fitness. Another reason why the BGA is more successful is that our genes are not limited. While changing a node connection may help provide a better solution, adding an additional node or connection may achieve a similar effect.

Future Work

While BGA was able to produce an agent with the ability to finish Donut Plains, no other level was tested. It may be possible that other levels, such as a level that requires the player to wait

or contain puzzles, may not be solvable with this solution. A possibility could be that on simpler levels, other GA variations perform better. Though a single level has been solved, the next step is to try and solve several levels at once. This could be done by testing the network on several levels and calculating the fitness with this in mind. It could also be done by having each member of the population do a randomly selected level to test their fitness on. The current Fitness calculations can be updated to support more complex behavior, such as having the agent try and finish the level as fast as possible, or try to discover secret sections of the level. The agent currently does not take into consideration what power up is equipped. Alterations could be added to the neural networks to only to have a new connection which is active when a specific power up is present. Additionally to increase the efficiency of every variation, a set of starting nodes and connections could be created to reduce having the algorithm generate nodes that have the agent perform essential actions, such as moving right or doing a longer jump at for higher platforms.

While BGA had the best performance for fitness, AGA creates simpler solutions. A new GA variation that takes aspects of both BGA and AGA may create an algorithm that not only has better performance, but generates simpler results. This could be greatly beneficial if each level must have a special created network to solve it, reducing the size needed to house all solutions. Additional GA variations such as GAVaPS and GPGA should be tested, to try and find a more successful variation than BGA. Additionally more trials should be conducted to help find the average fitness of each GA variation. For variations that did not solve the level, running them until it is solved(if possible) to better compare variations to one another.

Conclusion

From the experiments, Donut Plains was successfully completed through the use of a NEAT algorithm, using the BGA. BGA had higher average performance compared to the other tested variations. Having only five trials for each variation does not fully remove the chance of one algorithm generating a high quality solution initially. However, BGA never had a solution at the end of a trial within 500 fitness points of 2000 fitness, while all the other variations had multiple trials within that range. BGA has smaller plateaus of solution quality compared to the other variations. These effects are likely due to BGA only letting top members breed, removing poor genes that would override the usefulness of good genes. The other variations, GA, EGA and AGA all had similar solution averages, further providing evidence that BGA's higher performance is a result of its breeding selection.

The NEAT structure did not favor EGA, due to it always having a copy of the best solution. While adaptive presents itself as a solution if simpler networks are desired, or to reduce the consideration of mutation and crossover rates. A higher number of trials on other levels should be conducted by future work to find the solution averages between all variations. This will help reduce possible favoring of one algorithm due to the structure of the level. AGA, EGA and GA all contained trials that were near or exceeded 5000 fitness, demonstrating that more trials could show these variations are more viable than the data appears to demonstrate. The overall approach, regardless of GA variation had relative success, the agent performed actions similar to a human player, such as avoiding enemies or reducing speed when approaching more complex areas. Each variation contained at least one trial that had completed more than 50% of the level(fitness higher than 3500), had more generation been allowed other variation may have succeeded.

References

[1] Bizhawk. `http://tasvideos.org/BizHawk.html`. Accessed: 2018-12-12.

[2] Lua programming language. `https://www.lua.org/`. Accessed: 2018-12-12.

[3] Mari/o. `https://github.com/mam91/Neat-Genetic-Mario`. Accessed: 2018-12-12.

[4] Playstation 4 specs. `https://www.playstation.com/en-gb/explore/ps4/tech-specs/`. Accessed: 2018-12-12.

[5] Xbox one specs. `https://www.cnet.com/products/microsoft-xbox-one/specs/`. Accessed: 2018-12-12.

[6] J. Arabas, Z. Michalewicz, and J. Mulawka. Gavaps-a genetic algorithm with varying population size. In *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*, pages 73–78. IEEE, 1994.

[7] S. Baluja and R. Caruana. Removing the genetics from the standard genetic algorithm. In *Machine Learning Proceedings 1995*, pages 38–46. Elsevier, 1995.

[8] E. Cantú-Paz. A summary of research on parallel genetic algorithms. 1995.

[9] F. Gouidis, T. Patkos, G. Flouris, and D. Plexousakis. Dynamic repairing a*: a plan-repairing algorithm for dynamic domains. pages 363–370, 01 2018.

[10] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[11] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[12] Nintendo. Super mario world, 1990.

[13] D. Schlierkamp-Voosen and H. Mühlenbein. Predictive models for the breeder genetic algorithm. *Evolutionary Computation*, 1(1):25–49, 1993.

[14] M. Srinivas and L. M. Patnaik. Adaptive probabilities of crossover and mutation in genetic algorithms. *IEEE Transactions on Systems, Man, and Cybernetics*, 24(4):656–667, 1994.

[15] J. Tremblay. Improving behaviour and decision making for companions in modern digital games. In *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2013.

[16] C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[17] D. Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85, 1994.