Flap For AI

Authors:

Timothy Lang

Eddie Chen

Abstract

"Flappy Bird" is a game with an easy-to-understand goal: flap through a bunch of pipes with little openings between them to get points. While the game is easy to play, for many players, controlling the bird to go through the pipes is a different story. We will use a genetic algorithm called NEAT (NeuroEvolution of Augmenting Topologies) to control the bird in "Flappy Bird" to figure out what inputs are most useful in beating "Flappy Bird", and through numerous trials, we will find that only 3 inputs are necessary to solve "Flappy Bird". We will later optimize NEAT to more quickly produce these neural networks capable of beating "Flappy Bird". This is done by running a modified hill climbing algorithm to find the optimal parameters for NEAT and using them in an instance of NEAT. In half the time it takes an instance of NEAT using random parameters, this optimal instance of NEAT created a neural network that is capable of solving "Flappy Bird".

1 Introduction

"Flappy Bird" was a popular mobile game during 2014. A game with a simple premise of flapping through pipes to get points. While the game was on app stores, many players found the game to be really addictive, always reaching for their phones for the next game. Since the game

is so simple, losing really early on in the game frequently can get very frustrating. Instead of aimlessly playing the game, we decided it would be better for an AI to learn to play the game and teach us how to approach the game in a more systematic way.

Artificial Intelligence is a large field that started in 1950 when Alan Turing published his famous paper “Computer Machinery and Intelligence”. In his paper, he coined the term “artificial intelligence” and proposed a way to measure computer intelligence called “The Imitation Game”, which we know now as the Turing Test. Later in 1959, Arthur Samuel coined the term “machine learning” to describe the process of programming a computer to learn to play a better game of checkers than can be played by the person who wrote the program. While “machine learning” and “artificial intelligence” are 2 buzzwords thrown around in recent years that seem to be used interchangeably, they are two different fields. Machine learning is a field that falls under artificial intelligence. While artificial intelligence is the capability of a computer system to mimic human intelligence, such as using logic to simulate the reasoning that people use when making decisions based on known information, machine learning is the process of using data to help a computer learn without direct instructions. Under machine learning, there are 2 different goals: to learn something about a dataset or producing a result whether it be quantitative or categorical. In our case of solving “Flappy Bird”, the machine learning algorithm will produce an artifact that will be used to predict output results. Even if the machine learning algorithm is taking in a bunch of input, there are bound to be outliers that can throw off the model’s predictions. There are even cases where even if there are no outliers, there is a possibility that the output is wrong because the data the algorithm was trained on was randomly spread out. When data is randomly spread out, the algorithm can not easily find a relationship so no meaningful artifact can be produced. Keeping this in mind, we should always be thinking about the context

in which we use machine learning. Machine learning is often used when we do not know how to solve a problem. Some examples in which machine learning could be be used is creating a color using the select few colors available or classifying whether a robot should go forward, take a slight-left turn, take a slight-right turn or take a sharp-right turn when following a left wall, or classifying/deciding when an agent should flap depending on the environment's state and the agent's state.

To solve our problem of learning how to play "Flappy Bird" better, we decided to use an application of machine learning. The machine learning algorithm that we will be focusing on are neural networks that are usually used to model a human brain. In our case, we will be modeling a bird's brain. Going further than this, we will be using an algorithm called NeuroEvolution of Augmenting Topologies (NEAT) to build these neural networks from scratch. We will later on use a modified version of the hill-climbing algorithm to identify the ideal coefficients for NEAT to use when creating neural networks to solve the game of "Flappy Bird" and how these ideal coefficients doubled our speed in building a neural network that solves the game of "Flappy Bird".

2 Background

To understand NEAT, a good understanding of genetic algorithms and neural networks will be helpful. The goal of this section is to refamiliarize the reader with what these are and to provide details on how the NEAT algorithm works.

2.1 Genetic Algorithms

Genetic algorithms are algorithms that seek to find solutions to unconstrained and constrained optimization problems based on natural selection. To do so, genetic algorithms start with an initial population of randomly generated individuals which are usually represented as a set of properties that are known as the “chromosome” or “genotype”. These encodings are typically represented as binary strings. After setting up an initial population, these individuals are evaluated based on some objective, which is usually a measure that reflects how well the individual does at solving the given problem to be optimized. Then, the more fit individuals are selected from throughout the entire population to be parents for producing offspring of the next iteration of the population. We call the iterations of the population, generations. The offspring from a pair of parent individuals are usually a combination of the parents’ chromosomes and either have or do not have a mutation introduced in their chromosome. This cycle of evaluating individuals, selecting and crossing over chromosomes and then possibly adding on mutations to those chromosomes across different generations result in a solution to the unconstrained or constrained optimization problem.

2.2 Neural Networks

Neural networks are machine learning algorithms that fall under the category of supervised learning. This means that it is an algorithm that seeks to produce an artifact to predict an outcome. As mentioned earlier, neural networks are used to model the human brain. In the context of solving “Flappy Bird”, one neural network will be used to model one bird’s brain. Neural networks are made up of 3 types of layers: the input layer, the hidden layer(s), and the output layer. Within the input layer, only input nodes are allowed inside because this is the layer that is the initial point of entry for information from the environment. Each node of the input

layer represents different inputs from the environment. When the neural network first receives input from the environment, they are stored in the input layer nodes and then these inputs are transferred to nodes in the next layer, the hidden layer, via connections. These connections have weights applied onto them so the output of an input layer node is multiplied by this weight and is added onto the input value of the hidden layer node that the connection is directed at. After the hidden layer node adds up all the outputs received from the previous layer's nodes with the respective weight applied onto each output, a bias, which is attached to the receiving node, is added to produce an output for this node. After this calculation, this value is passed as an argument to the activation function, and what is returned by the activation function is the output of the node.

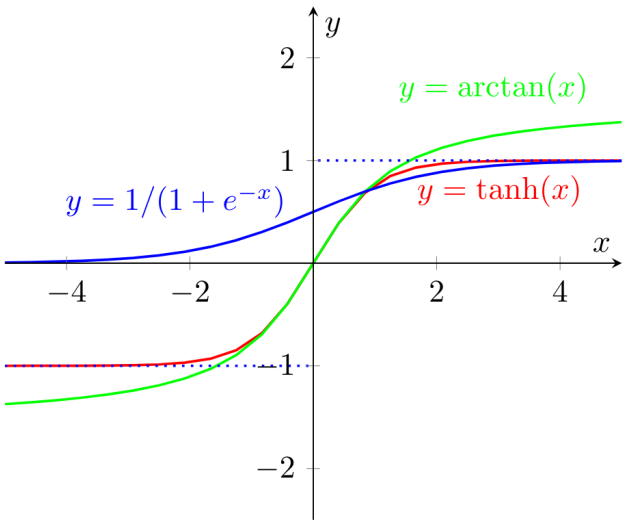

Figure 1: A picture of some common activation functions: the Sigmoid Function (in blue), the Hyperbolic Tangent Function (in red), and the Inverse Tangent Function (in green) (Oken, n.d., p. 4, fig. 5).

This pattern of taking the previous layer's nodes' outputs, applying the weight to them according to the connection it came through, adding those outputs and the corresponding node's bias together, and passing them as an argument for the selected activation function to become the output of a node will continue all the way to the output layer. The output value produced by the nodes in the output layer will determine whether or not a particular decision will be made. What determines this final decision is often based on the context of the problem. After this forward-pass, the neural network's connections' weights and the neural network's nodes' bias are typically updated through gradient descent of an error function, which is a measure of how far off the neural network's output is from the target output. By doing all this, the neural network simulates how a brain would learn how to do something.

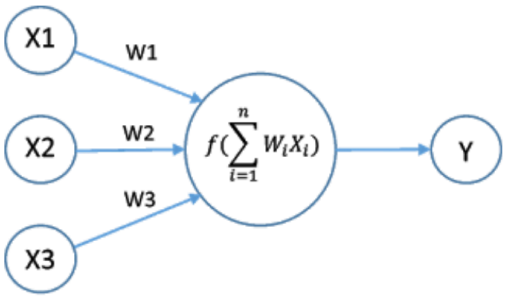

Figure 2: A visual for how a node receives input and calculates output, where "X1", "X2", "X3" are inputs, "W1", "W2", "W3" are weights attached to the connection, and f represents the activation function being used (Oken, n.d., p. 2, fig. 3).

2.3 NEAT

With this brief understanding of genetic algorithms and neural networks, we will break down what NEAT is and how it works. NEAT is a genetic algorithm and a TWEANN (Topology

and Weight Evolving Artificial Neural Network algorithm). The goal of NEAT is to use natural selection to evolve the smallest and most efficient neural network for a given problem.

2.3.1 Initial Population and Genetic Encoding

As a genetic algorithm, there needs to be some initial population to work with. In the context of our problem, the neural networks that constitute the initial population represent the brains of our birds, or more simply, the bird. Now that we know what makes up the initial population, Stanley and Miikkulainen (2002), the researchers who developed NEAT, propose that we encode individuals using genomes (p.107). A single genome would then represent a neural network that acts as a bird's brain. The genome is made of 2 lists. One list to keep track of the nodes in the neural network and another list to keep track of the connections in the neural network.

Though the example in Figure 3 is one where the neural network has connections and a hidden layer node, NEAT starts off with the most minimalist structure possible: an empty neural network which does not contain any hidden layers or connections i.e there are only input layer nodes and output layer nodes. This is to ensure that the smallest possible search space is formed throughout the iterations of NEAT, which means the final solution neural network produced is the most efficient neural network for the given problem. After encoding the individuals and setting up the initial population, the individuals must be evaluated in some way to get a measure of fitness. This will typically depend on the context of the given problem that the neural network is working to solve. Once the entire population is evaluated, they will be speciated.

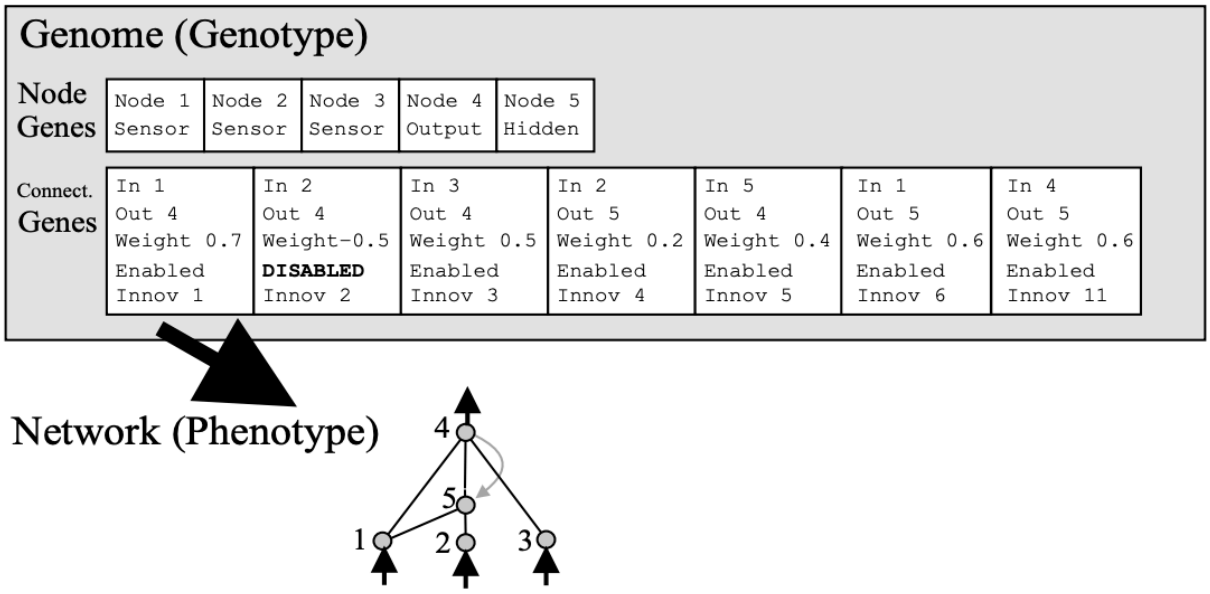

Figure 3: An example of how the genome is represented and the physical representation of the genome as a neural network (Stanley & Miikkulainen, 2002, p.106, fig. 2).

2.3.2 Speciation

Most of the time when TWEANNs introduce a structural innovation, the fitness of the neural network initially drops. This causes other TWEANNs to discard these structural innovations before giving them time to develop. With NEAT, these structural innovations are protected through speciation. By putting these neural networks into species, the neural networks have time to develop these structural innovations within the species before competing in the larger population.

To perform speciation, there needs to be a way to compare two genomes to each other, particularly the genome representing a species and the genome that does not yet belong in a species. To do so, knowing what makes 2 genomes different from each other is important.

Stanley and Miikkulainen (2002) set the criteria as the number of excess genes, the number of disjoint genes, and the average weight differences of the matching genes (p. 110). Excess, disjoint, and matching genes are the result of comparing two genomes. Notice how in Figure 3, in the list of connections in the genomes, all the connections have something labeled "Innov 1", "Innov 2", etc. These are called global innovation markers and they provide information on when a connection gene was introduced into the genome. Using these innovation markers, all genomes can be crossed over, even if the genomes are of differing sizes. If the innovation numbers are matching, then we have what is known as a matching connection gene. In the case that the innovation numbers are not matching and one genome has an innovation number the other genome does not have, those genes are considered disjoint or excess genes. The difference between them is that a disjoint gene is a gene that has an innovation number between the innovation numbers in the other genome and an excess gene is a gene that has an innovation number exceeding the maximum innovation number in the other genome.

After establishing this criteria, a compatibility function can be created from this criteria to distinguish whether a neural network belongs in a particular species or not. In the compatibility function shown in Figure 4, δ represents the difference score between two genomes, "E" represents the number of excess genes, "D" represents the number disjoint genes, "$\overline{W}$" represents the average weight differences of the matching genes, and "N" represents the total number of genes in the larger genome. The coefficients c_1, c_2, and c_3 are used to adjust the importance of these genes in calculating how different 2 genomes are.

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \cdot \overline{W}$$

Figure 4: The compatibility function (Stanley & Miikkulainen, 2002, p. 110, equation 1).

Now that the compatibility function is established, a compatibility threshold needs to be set depending on the problem at hand. If the difference score between the representative of a species and an individual is lower than the threshold, then the 2 individuals are said to be similar and should be put in the same species. Otherwise, they are not considered to be of the same species so the individual is tested against representatives of other species and if this individual does not join another species, then it is designated as the representative of a new species.

2.3.3 Crossing Over

After the speciation process is finished, parents are selected from within their respective species and then crossed over to produce a certain number of offspring per species.

The first step in crossing over is to remove the least fit individuals from the population. After doing so, the parents are selected from within the species to cross over. (The method of parent selection is not explicitly mentioned by Stanley and Miikkulainen's paper so we assume that this is up to the interpretation of whomever wants to implement NEAT.) Their genomes are then matched up to each other. For matching genes, the weight of the connection gene in the offspring is chosen randomly from either parent. When it comes to disjoint and excess genes, only the disjoint and excess genes from the more fit parent are passed on to the offspring.

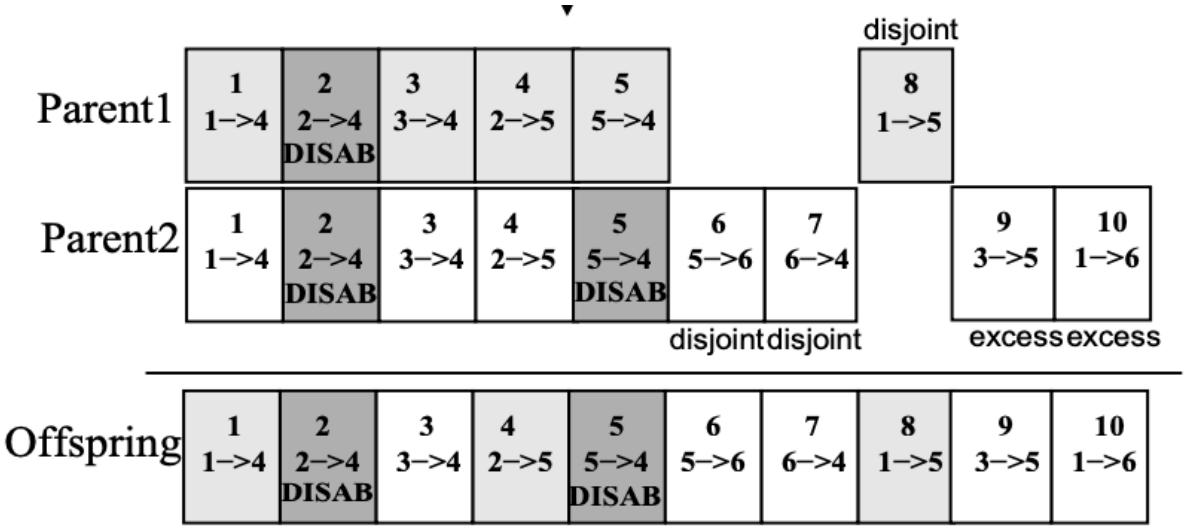

Figure 5: An example of what crossing over looks like under the condition that both parents are equally fit (Stanley & Miikkulainen, 2002, p. 109, fig. 4).

This process of crossing over within the species will stop once the population is filled back up to the initial size of the population. Each species has an allotted number of offspring that they can produce. This allotted number of offspring that a species can have is determined by explicit fitness sharing. Explicit fitness sharing is when organisms of the same species must share the fitness of their species. To do this, the adjusted fitness score of an entire species needs to be calculated.

The adjusted fitness score formula shown in Figure 6 will take the fitness of an individual and divide it by however many individuals are compatible with it. After finding the adjusted fitnesses of all individuals in a species, the adjusted fitness scores are all added up for that species. The number of offspring that a species can have will be in proportion to this sum and the total adjusted fitness of the entire population.

$$f'_i = \frac{f_i}{\sum_{j=1}^{n} \mathrm{sh}(\delta(i,j))}$$

Figure 6: Adjusted Fitness Score formula, where i represents the individual i, j represents every other organism, f'_i represents the adjusted fitness score of an individual, f_i represents the fitness score of an individual, and the function "sh" represents the logical sharing function which returns 1 when the individuals i and j are compatible and 0 when they're not compatible (Stanley & Miikkulainen, 2002, p. 110, equation 2).

2.3.4 Mutation

After crossing over the parent genomes, the offspring produced has a chance of incurring a mutation. Since the offspring are neural networks, there are 3 ways they can mutate: adding a connection between two nodes, adding a node, and mutating the weight of a connection (Stanley & Miikkulainen, 2002, pp. 107–108).

When it comes to adding a connection, it is formed between two nodes that were unconnected and then a random weight is attached to that connection. When adding a node, an existing connection between two nodes is broken and a connection from the previous starting node is directed at the newly added node and a connection from the newly added node is directed at the previous ending node. The connection directed at the newly added node takes on a weight of 1.0, and the connection leading out of the newly added node takes on the weight of the connection that previously connected the starting and ending node. By doing so, the newly added

structural innovation has not harmed the fitness of this network. Furthermore, with the help of speciation, this newly added structural innovation will then be given time to develop.

All these steps starting from the initial population, to evaluating them, to speciating them, crossing individuals over, and applying a possible mutation are what make NEAT a genetic algorithm that produces the most efficient neural network for a given problem.

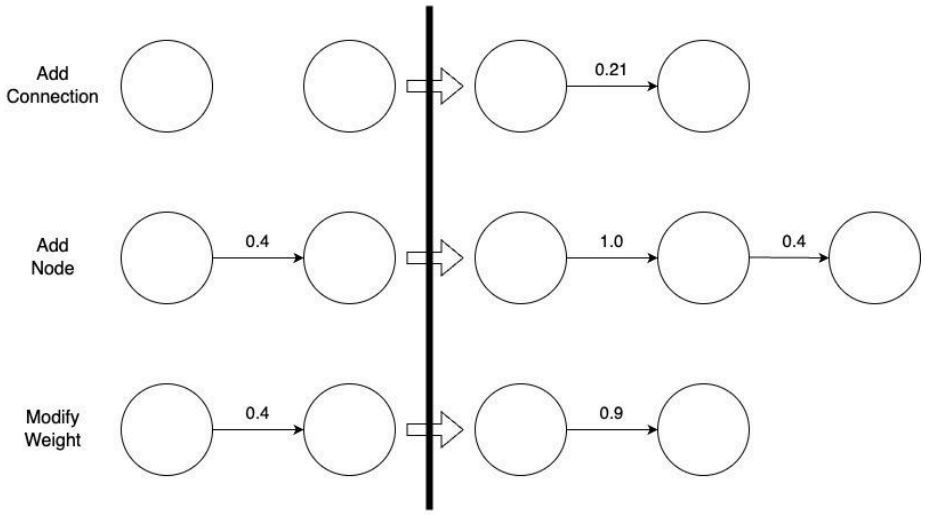

Figure 7: Example of mutations that can occur to a neural network.

3 Related Work

Before moving on to how we implemented the NEAT algorithm to solve “Flappy Bird”, we would like to give a brief mention to the YouTube channel, Code Bullet, who not only inspired our use of NEAT, but also gave critical evidence in how a node can connect to 2 different layers at once and not just to the layer immediately next to the layer the node the connection is coming out from.

4 Implementation

In this section, we will explain key concepts and data structures we used to implement NEAT.

4.1 Helper Classes

We initially started with the most basic classes which included the gene, neuron, and the activation function class. Since each gene represents a connection, we concluded that the most important information to be stored was the start neuron identifier, the end neuron identifier, and the weight of the connection. For the neuron, we stored its activation function inside it and represented its identifier using an integer. We created the activation function class as a simple interface to allow easy swapping between different functions we might use such as ReLU or sigmoid. In the end, we chose ReLU for the activation function because it was the least computationally intensive activation function we could find, and it proved to work in solving "Flappy Bird".

4.2 Genome Class

The first part we chose to implement was the genome class. The genome class contains a list of the neurons as well as a gene dictionary mapping starting node identifiers to a list of genes where genes represent the connections within the neural network. The gene class will hold only two data pieces: the end node identifier and the weight. We chose this implementation of the genome class where the genes contain the end node identifier and the weight over having a list of genes where each gene holds the start node identifier, the end node identifier, and the weight because we wanted to see what a neuron was connected to in O(1) time. We'll use this concept

later, but for now, we initialize the gene dictionary to be empty which means our neural networks all start with no connections.

As for the list of neurons, we initialize the input neurons and the output neurons. For example, if there are three inputs and two outputs (as with our final neural network), which we declare as constants for our neural networks, our initial list would have five neurons.

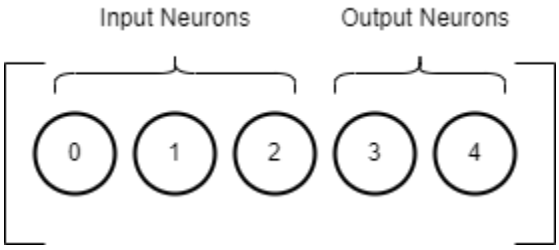

Figure 8: Initialization of the neuron list.

4.2.1 Mutations Within Genome

In our next step, we implemented the mutation process of the genome which includes modifying weights, adding connections, and adding nodes.

To modify the weight, we would pick one random starting node identifier from our gene dictionary, then pick one random gene from the list of genes and modify its weight by $\pm \frac{Diff}{2}$, where *Diff* is the *Differential* constant in our Neural_Constants file. *Differential* represents the maximum range we want to initialize new connection weights as well as the range of change, centered at 0, those connections can be modified by, during a mutation.

To add a new connection, first we create a new dictionary where the key is a starting node identifier and the value is a list of the identifiers for all the possible nodes the starting node can connect to. The rules for possible connection nodes include: the end node's index has to be higher than the starting node's index in the list of neurons, input layer nodes can't connect to

each other, and output layer nodes can't connect to anything. This indexing rule ensures that there are no cycles within the neural network because the connections can only move forward along the list, not backwards. Input layer nodes should not be able to connect to each other because they are the starting points for the neural network. Output layer nodes don't connect to anything because these nodes are in the final layer and should be giving us an output. After creating this new dictionary, we pick a random starting and ending node from the remaining possible connections and establish a connection between them by updating the existing gene dictionary. When the connection is established, a weight between $\pm \frac{Diff}{2}$ is then attached to it.

We add a new node by breaking an existing connection between 2 nodes i.e inserting a new node in between them, connecting the starting node to the newly added node, and connecting the newly added node to the ending node (Stanley & Miikkulainen, 2002, pp. 107–108). The identifier of the new neuron will be equal to the length of the neuron list. The connection between the starting node to the new node will have a weight of one whereas the connection between the new node to the ending node would retain the original weight that the former connection had (Stanley & Miikkulainen, 2002, pp. 107–108). This means that adding a new node will initially have no effect on the output of the neural network since the original weight was maintained. With this in mind, we insert the node into the node/neuron list with the following rules: its index has to be higher than the starting node, but less than the index of the ending node of the connection it's breaking, and it can't be inserted between any input layer nodes nor can it be inserted between any output layer nodes. This insertion practice ensures no cycles will occur because the flow of connections is maintained from left to right.

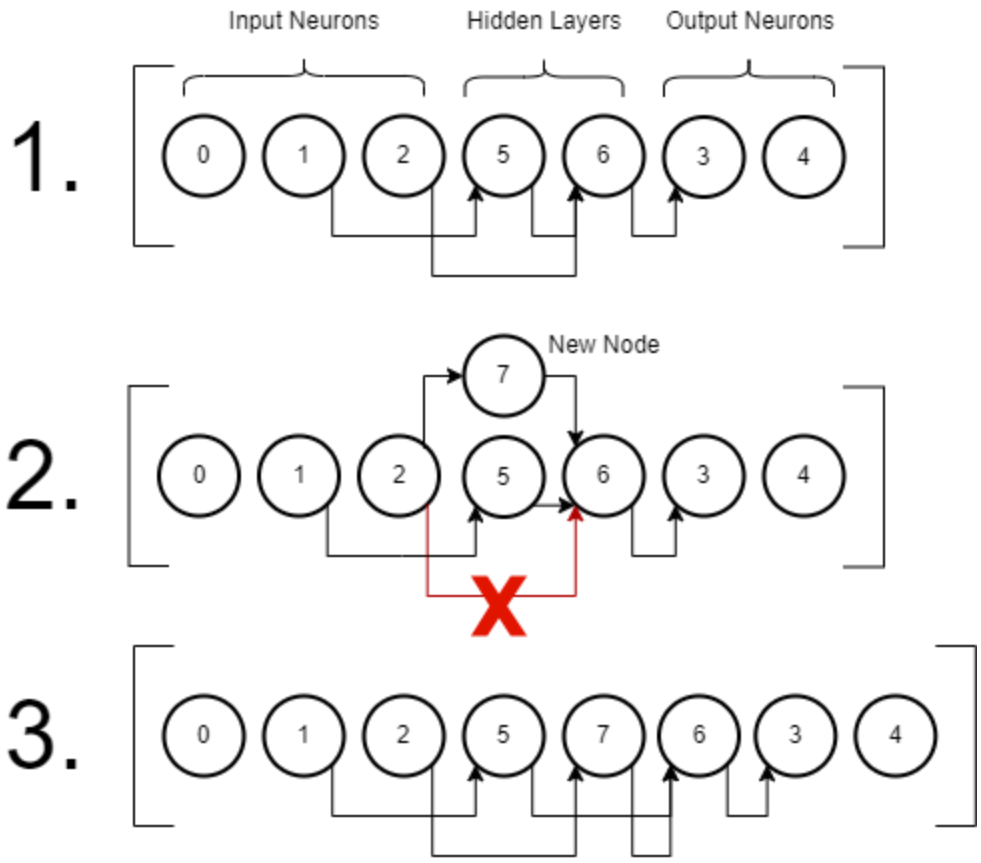

Figure 9: Inserting a neuron into the neuron list.

4.2.2 Crossover Within Genome

The next method that we implemented was the crossover of 2 genomes. Unlike the research paper by Stanley and Miikkulainen, we decided to not include the global innovation marker feature because we realized that we could identify every connection by comparing the start and end node identifiers. If two different genomes shared the same connection between the same two nodes, then we have a matching gene. In the case of a matching gene, we could pick this gene randomly from either parent genome to include in the child. However, in going through without the innovation marker, we ran into a small problem. Since we insert nodes randomly into

the list (as long as the flow of connections is maintained left to right), we might end up with the following scenario shown in Figure 10:

There are two possible resulting neural networks that have identical structures, meaning that if we removed the node identifiers, they would look the same. However, when different node identifiers are used in these identical structures, the only genes that would get used during the crossover process is the connection between node two and node seven since it is the only connection with matching input and output identifiers; none of the papers we have read address this issue nor acknowledge it. After testing, printing, and comparing neural networks, we found out this situation occurs infrequently. We believe the primary reason is if two neural networks have enough disjoint genes, then they would end up in different species. If they end up in a different species, then there would be no situation where these genomes are crossed over since the process of crossing over only applies to genomes within the same species. While we acknowledge this is a minor issue within our code, it didn't stop the neural networks produced by the NEAT algorithm from completing "Flappy Bird". That combined with the fact that no other research paper addressed this issue with their implementation outline also indicated that having two species with similar structures isn't a major problem and so we didn't pursue it any further.

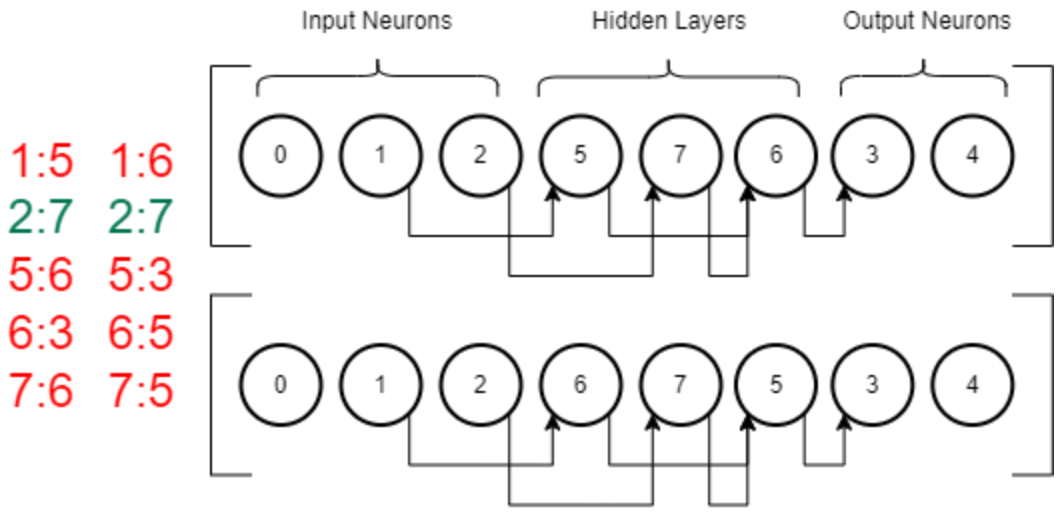

Figure 10: Example of two structures that are identical, but wouldn't end up in the same species.

The last major method we implemented in the genome class was the output method which takes in the inputs and calculates the values for the two output layer nodes once all the inputs have been processed by the neural network. To begin with, we start with the first node and work through the neuron list sequentially. For each node, we check to see if that node connects to another node by looking them up in our gene dictionary. For each connection:

1. Take the value stored at the current node
2. Multiply the value by the weight of the connection
3. Run it through the activation function associated with that node
4. Send the new value to the respective output node of the connection

Both the end node and weight are stored in each connection gene. As a result of our earlier insertion practice, the end nodes must always have a higher index in the list, so when we go through this list sequentially, by the time any node in the list is reached, it should have all the inputs that it needs to calculate its output.

4.3 Species

The next class we created was the species class. The first task we solved was figuring out if two birds belong in the same species. We decided that we would have one neural network (which we call the basis) be the representative of the species. Every new bird will need to be compared with this representative to join the species. This strategy was also suggested by Stanley and Miikkulainen (2002) and comes with the advantage of being in constant time (p. 110). With this in mind, our similarity function checks to see if a bird is similar to the basis/representative of the species instead of checking to see if a bird is similar to the average bird in the species. If the similarity function determines that the bird is similar to the basis of the species, then that bird becomes a part of that species. If the bird is not similar to any representative of the species in the remaining list of species, then this unique bird is marked as being a part of a new species and is set as the representative of this new species.

The other important method we implemented was creating the offspring within a species. To calculate the number of offspring, we would first calculate how similar each bird is to every other bird in the population, and adjust the score of the birds to emphasize unique birds over common birds using the explicit sharing function. Using these adjusted scores, each species would take the sum of those scores, normalize them, multiply by the total population we want for the next generation, and that would result in the total number of offspring for each species. Then, we used a roulette selection to pick the parents for the next generation. Roulette works as the following: each bird has a percent chance to get picked as a parent based on its adjusted fitness score. The higher the score, the more likely that genome will be chosen as a parent for the next generation. We then pick x parents, where x is the number of offspring that a species can

produce. After the parents are selected, they are paired up and crossed over to produce two offspring per pair to ensure we hit the target x amount of children. One special case we took into consideration was if there was an odd number of parents, then the final parent wouldn't be paired with any of the other parents and would by default be added to the list of offspring for the next generation. We then apply mutations to the offspring. Finally, the species would then return the offspring. This species would not be used again as each generation would produce its own unique species.

4.4 NEAT Class

With our basic classes setup, we then turned to implementing NEAT itself. On creation, NEAT would create a number of birds equal to the population constant within Neural_Constants. Each bird would start with a genome that would just be an empty neural network apart from the initialized input and output nodes. Our code would continually loop until a single bird hit the max score which we defined within Neural_Constants. While looping, our program would do the following:

1. Run the simulation until all birds die or one reaches the max score
2. Assign birds into their appropriate species
3. Adjust bird scores to promote unique birds using the explicit fitness sharing function
4. Calculate the number of offspring each species needs to produce
5. Create the offspring
6. Repeat using the offspring as the new population

The final step was to figure out what inputs and outputs we needed. For the outputs, we decided to have two output nodes (labeled 9, 10) where the first output node represents the bird jumping and the bottom node represents the bird not jumping. If the first output node has a greater number than the bottom node, then the bird will jump at that frame. In all other cases, the bird will choose not to jump. For the input nodes, we tested across a variety of inputs including:

1. The bird's y location
2. The lower pipe's y location
3. The upper pipe's y location
4. The next set of pipes lower pipe's y location
5. The next set of pipes upper pipe's y location
6. The velocity of the bird
7. The distance between the bird and the edge of the first set of pipes
8. The distance between the bird and the edge of the next set of pipes
9. Output node telling the bird to jump
10. Output node telling the bird not to jump

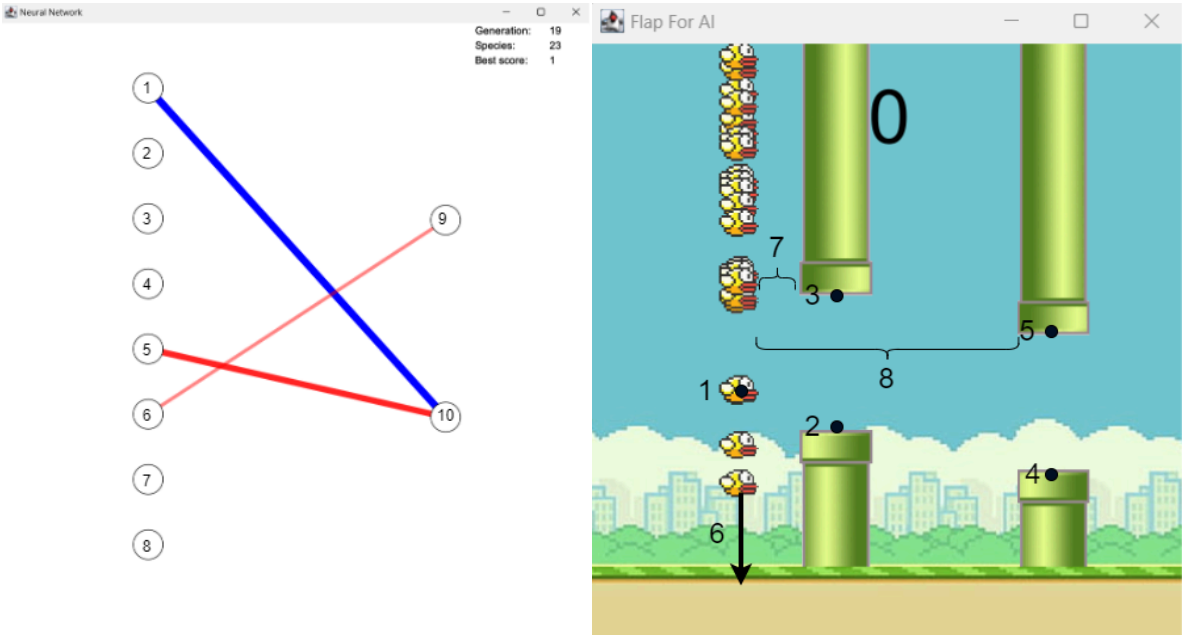

Figure 11: (On Left) Example neural network with labeled inputs and outputs

(On Right) Diagram showcasing inputs.

After thorough testing, we decided on using the first three inputs only because our AI wasn't able to solve Flappy Bird without these three inputs. As for the other five, we experimented with removing and adding in different combinations and concluded that none of them were necessary nor helped improve the speed of the neural network learning Flappy Bird.

5 Analysis

After implementing NEAT, our next goal was to improve it to solve "Flappy Bird" in the least number of generations, where we consider the empty neural network generation as generation 0.

5.1 Coefficients

Our first step was to identify which coefficients in the NEAT algorithm we needed to optimize to lower the number of generations it would take for it to solve “Flappy Bird”. In the end, we chose six coefficients to adjust:

1. Chances of adding a connection during a mutation
2. Chances of adding a new node during a mutation
3. Chances of modifying a weight during a mutation
4. Differential: as mentioned previously, the differential is the range that a connection weight can start and also represents the range at which a connection weight can change during modifying weight mutation.
5. Average weight coefficient: coefficient used in calculation to determine if two birds are similar and should belong in the same species. It is multiplied by the average of the difference between all matching genomes’ weights when comparing two genomes to each other (Stanley & Miikkulainen, 2002, p. 110).
6. Excess-disjoint coefficient: we decided to combine the excess coefficient with the disjoint coefficient to lessen the total number of coefficients we needed to test. This coefficient is used the same way the average weight coefficient is except that this coefficient is multiplied by the number of genes that don’t match up when comparing two genomes, so in this case, all excess and disjoint genes will be treated the same.

We wanted to limit the total number of coefficients as much as possible due to time restrictions. The less coefficients we were changing, the more time we could dedicate to figuring out what the ideal value is for the coefficients we are changing because we have less variables to

change. Therefore, here are some of the other important coefficients we declared as constants. This list also includes other important parameters we set:

1. Total population of birds: we assumed that the more birds we have, the faster NEAT will solve "Flappy Bird" because it has more variations within each generation, so it has a higher chance of getting "lucky" with one variation that mutated just right. We set this to an arbitrary number of 100 birds.
2. Activation functions: we decided for all middle nodes to use the ReLU activation function, and all our input and output nodes use the identity activation function.
3. Max score: after careful observation, we found that with the three inputs, the majority of birds that make it to 200 pipes rarely died past that, so we capped the maximum score at 200 at which point the simulation would finish.
4. Difference threshold: the threshold that determines if two birds belong in the same species or not. If δ represents the difference threshold, c_1 represents the excess-disjoint coefficient, ED represents the number of excess and disjoint genes, c_2 represents the average weight coefficient, and $\overline{W}$ represents the average weight value, then when $\delta > c_1 * ED + c_2 * \overline{W}$, the two birds will be in the same species. We decided for this to be a constant because changing this variable would effectively be the same as modifying both the average weight and the excess-disjoint coefficient. For example, increasing the difference threshold would have the same effect as lowering either the average weight or excess-disjoint coefficient: in both cases, we would have less total species. In the end, we arbitrarily set the difference threshold to one.

5. Max generations: we set the maximum generations before the simulation quits to be 30 since based on our initial data, only 5% of the simulations reached beyond 30 generations. If any simulation reached 30 generations without solving “Flappy Bird”, we would say that it did solve it at 30 generations and end the simulation since we would gain very little information at a high cost of time (sometimes double the length of the run if we’re unlucky enough) past the 30 simulations.

5.2 Testing

We then randomized the coefficients over 3000 trials to see what the average number of generations NEAT took to solve “Flappy Bird” as a starting point for our analysis. In Figure 12, all the trials below 17 generations are displayed for easier comparison with our final output.

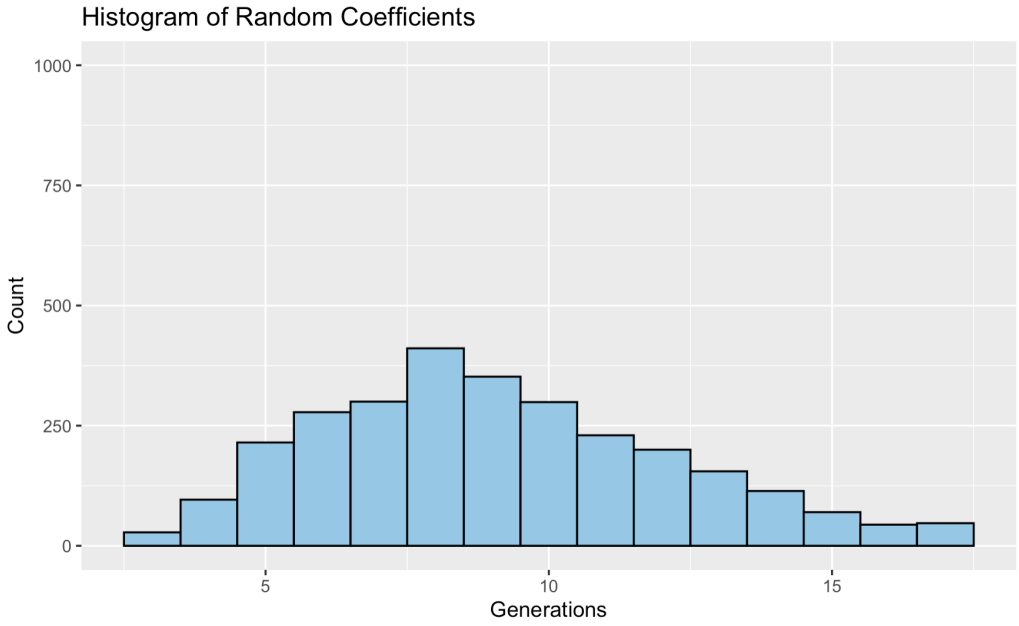

Figure 12: A histogram displaying the number of generations that instances of NEAT using random coefficients took to solve “Flappy Bird”.

We had a starting average of 10.08 generations to solve “Flappy Bird”. To improve our average generations, we created a hill climbing search with multithreading to speed up the process of finding the ideal coefficients. With our six changing coefficients, we looked at the 12 neighbors of our starting neural constants and picked the best neighbor. Each neighbor is a single coefficient from the set of the changing coefficients we were working with ±0.05. There were some neighbors that were excluded if they ever came up:

1. If the value of a probability (ex. add connection) is higher than one, we wouldn’t consider that neighbor since it makes no difference whether it’s 100% of a mutation or 200% chance of a mutation.
2. Add connection or differential can’t drop to zero. For adding connections, since we start with an empty neural network, if we had a 0% chance to add a new connection, we wouldn’t get anywhere. Differential is used as a range to initialize the connection, so if the differential is zero, then it would be the same as adding a connection with a weight of zero. That would be equivalent to not adding any connections at all.
3. The excess-disjoint coefficient can’t rise above one. Since the the difference threshold is set at one, then if there was a single disjoint gene, it would automatically put the bird in a new species because one excess gene * excess-disjoint coefficient of one >= difference threshold of one. If the coefficient were any higher, it would have the same effect of automatically adding the bird in a new species, so we decided to cap it at one.
4. None of the parameters can drop below zero, either because they represent a probability or it would be counterproductive to test below zero.

With these rules set in place, we could reduce the number of neighbors we would need to test with certain coefficients, thus speeding up our testing. Our next task was to decide on the parameters for our simulation. We decided to run 2000 trials for each neighbor, where in each trial ended whenever either a maximum score of 200 is reached or 30 generations is reached, whichever came first. We chose 2000 trials since it resulted in reasonable average generations and due to time constraints; We chose 30 generations because at that point that trial could take up too much time. From there, we would pick the best neighbor of the 12 or less neighbors relative to the starting state, and run the same process with all the neighbors of that neighbor and so forth until none of the neighbors were better. One issue with this method was the range of average generations we got back with just 2000 trials. We quickly realized that with such small changes in the parameters, it would require far more trials to get an accurate representation of whether the change had a positive or negative impact. For example, if we adjusted the excess-disjoint coefficient by +0.05, this new modification would result in a range of average generations instead of a solid consistent average generations where a portion of the range, say 60% of the time, it was better than the original coefficients, but the other 40% of the time, it was worse. Therefore, the results would all have similar trends, but would end before they got anywhere conclusive because all the neighbors were worse due to unfortunate circumstances of our limited number of trials.

For more substantial and informative results, we changed our hill climbing algorithm to pick the neighbor with the lowest average number of generations instead of comparing the neighbors to the original state's average number of generations. Since there's no stopping condition, the simulation would have no end and would continue to run as long as we wanted.

We hypothesized that, given enough time, it would converge to a local or global minimum average number of generations to complete “Flappy Bird”.

With the parameters and the code set, we chose a seven hour period to run our hill climbing on eight different computers simultaneously. During the seven hours, we found the average number of steps each computer took in its hill climbing was 75. Each computer tested ~10 neighbors at a time, each neighbor being run for 2000 trials for an average number of generations it took for those set of NEAT coefficients to solve “Flappy Bird”. Therefore, we estimate the total number of individual trials we ran to be about 12 million trials:

$$\frac{2000\ trials}{1\ neighbor} * \frac{\sim 10\ neighbors}{1\ step} * \frac{\sim 75\ steps}{1\ simulation} * 8\ total\ simulations\ =\ \sim 12,000,000\ trials \qquad (1)$$

5.3 Results

We graphed our end result, showcasing the average of the coefficients across all eight computers (Figure 13) as well as the average number of generations (Figure 14). We chose to limit the number of steps shown to be 65 because past 65 steps, the data becomes more jagged and the trends less clear. This is the case because past 65 steps, we look more at the actual coefficients rather than the average of the coefficients as more and more simulations end. This leaves fewer data points to the point that by the last 20 steps, we would be looking at the actual coefficients of Computer 5 which had 107 total steps, the most steps completed within the 6 hour period.

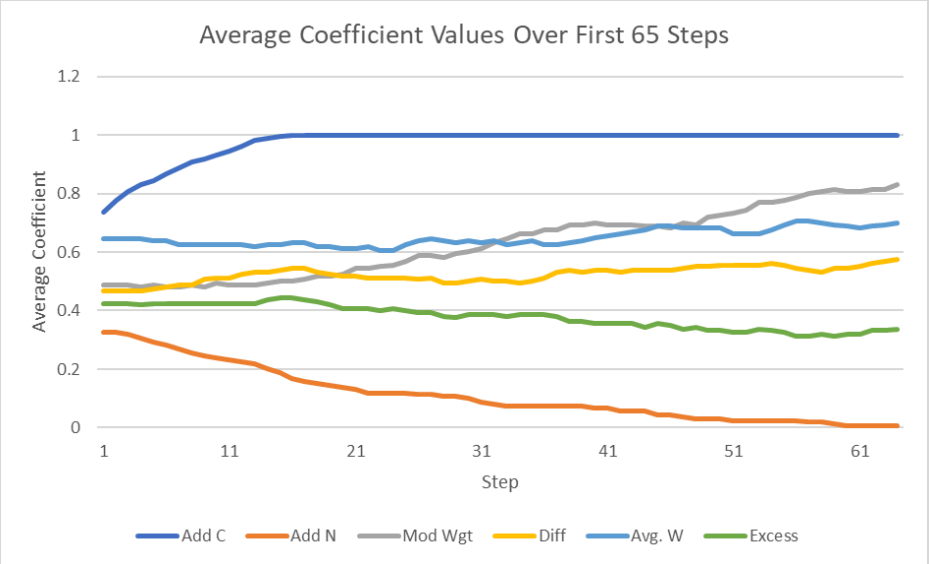

Figure 13: Average coefficient values using eight hill climbing simulations.

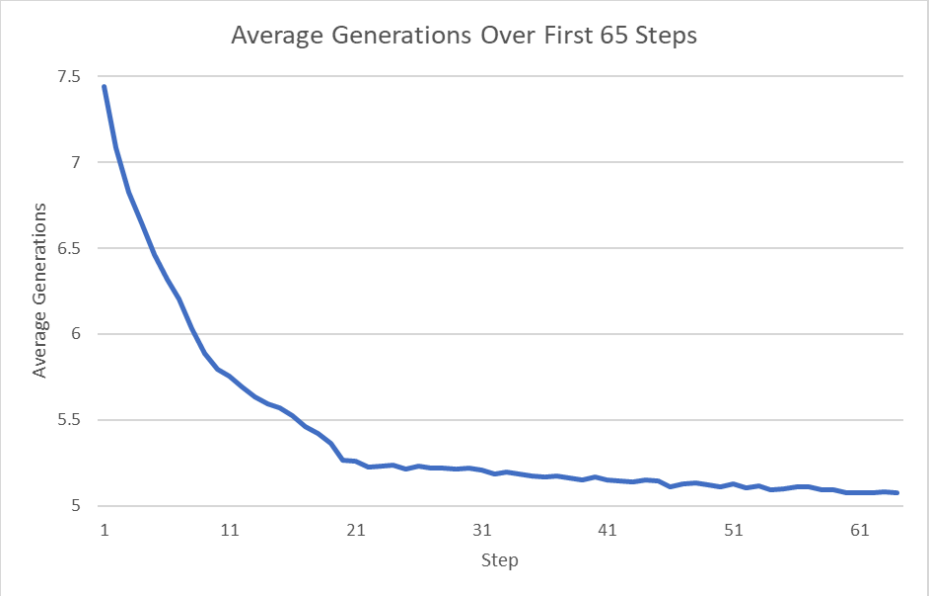

Figure 14: Average number of generations of all simulations over first 65 steps of hill climbing algorithm.

We will do a step-by-step breakdown of each coefficient in order of importance and our theory as to why it acted as such:

1. Add connection: for every simulation, the most dramatic improvements to the average number of generations to solve “Flappy Bird” was by increasing the chances of adding a connection into the neural network. In every single simulation, the first coefficient to

change was “add connection”, and it would continually climb until it reached 100% chance. This meant that for every generation, the children of that generation would add in a connection automatically, if possible. We believe the reason for this is because we gave the neural network inputs we suspected it would require to complete “Flappy Bird”; Therefore, since the neural networks all start empty, the best neural networks are the ones that can incorporate those inputs as fast as possible. The fastest and best way to incorporate these inputs was to connect them to the output nodes, and so the add connection coefficient spiked to one.

2. Add node: this coefficient was more subtle than the “add connection” coefficient as it took a fairly long time to drop, but it did so at a steady rate. If we were to continue this graph, we would find that the “add node” coefficient drops to zero in all cases. We believe the reasoning behind this is because “Flappy Bird” is so simple so it doesn’t require a complex neural network, just a basic one where the inputs connect directly to the outputs. If we have nodes in the middle, then there is a chance that the add connection mutation will add one from an input node to a middle node rather than directly to an output node. This in turn could potentially require an extra generation for the children to add in the correct connection.
3. Mod weight: the chance of modifying a weight during a mutation also increased across all simulations regardless of starting point. Seven out of the eight simulations ended with a mod weight chance of 85% or higher while the last simulation had a “mod weight” coefficient of 40%, but was last seen climbing at a slow rate before we ended it. Our theory is that once the connections are all established, all that was left of the neural network to do was to modify the weight until it got it exactly right.

4. Diff: the differential had a tendency to oscillate, but we were unable to draw any conclusions about the ideal differential. The only consistency we were able to derive was that seven out of the eight differentials had higher ending values than their starting values; however, due to the wide range of ending values, we don't believe that the size of the differential matters. Whether the range of the starting connection weights is between -0.5 to 0.5 or -2.5 to 2.5, it did not affect the NEAT algorithm's performance.

For the final two coefficients, we suspected that there may be a correlation between the two since they belonged to the same equation. Below in Figure 15, we have the ending values of the coefficients:

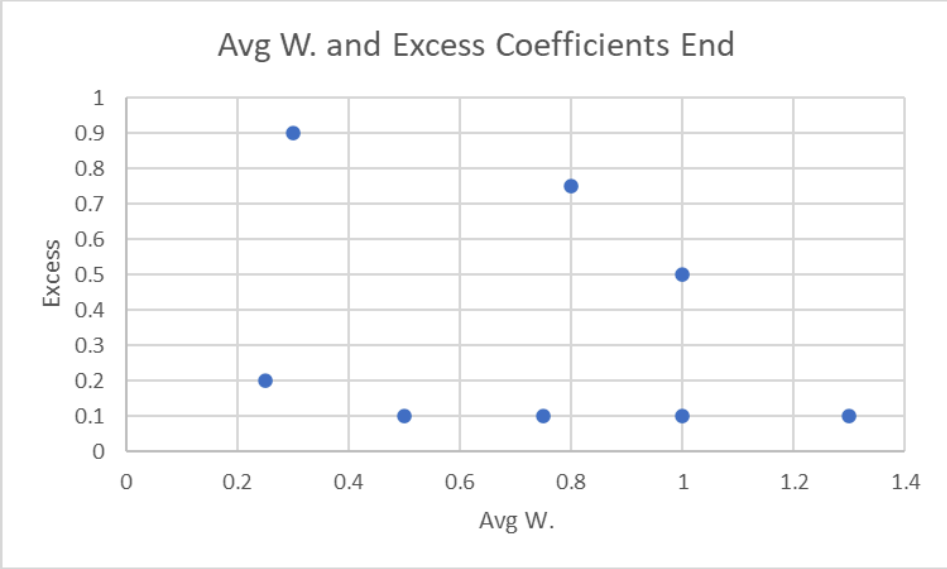

Figure 15: Average weight and excess coefficient ending relationships.

We believe that the ideal value for the excess-disjoint coefficient is 0.1 considering four of the eight simulations ended on that value; however, we were unable to determine a correlation between the two coefficients. We were also unable to determine an ideal value for the average weight coefficient because of its large spread, with values ranging from 0.25 to 1.3. Due to the

wide range of values that both coefficients had, we believe that the excess-disjoint and average weight coefficient values had very little effect on the behavior of NEAT. This led us to believe that speciation was not necessary to produce neural networks to solve a simple game like "Flappy Bird". Whether it had a high number of species or just one species throughout, it was able to solve the game within roughly around five generations.

The coefficients with the best score out of the 8 simulations, an average of 4.945 generations to solve "Flappy Bird", had an excess-disjoint coefficient of 0.1 and an average weight coefficient of 0.75. For our final set of ideal coefficients, we chose 0.1 for the excess-disjoint coefficient because it was the most common ending point. We then chose 0.5 for the average weight coefficient and 1.0 for the differential coefficient by taking the average ending value and accounting for their general trend.

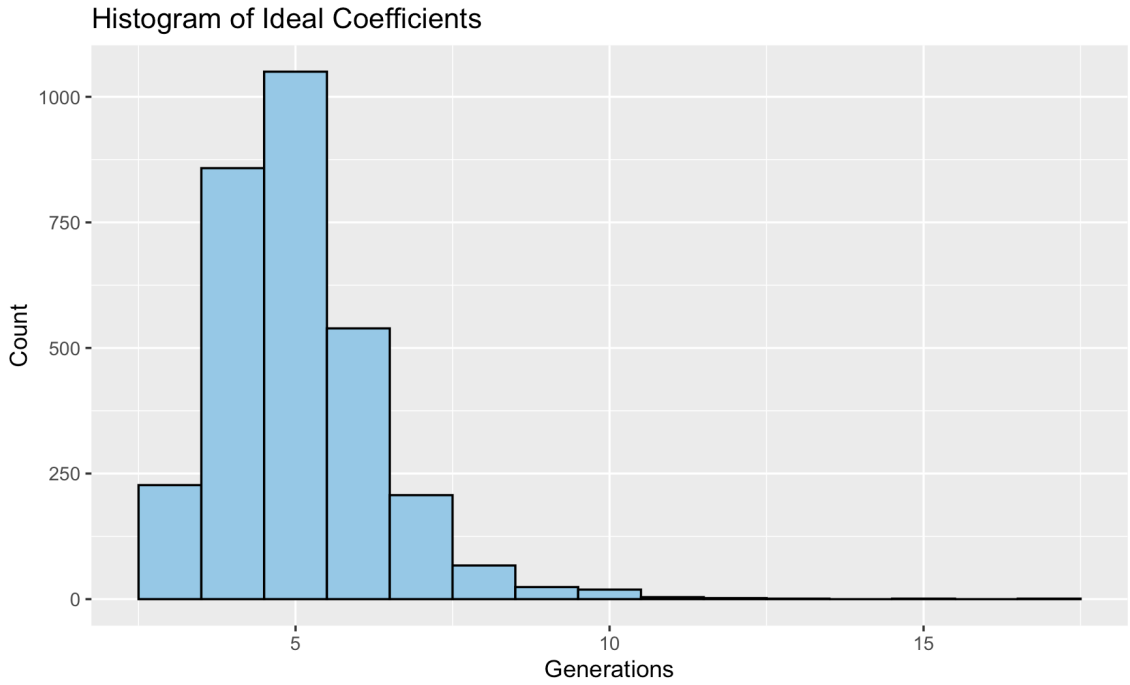

Figure 16: Histogram containing 3000 trials of the number of generations to solve "Flappy Bird" when NEAT uses the ideal coefficients.

5.4 Testing Resulting Coefficients

In the end, we chose the following set of coefficients: {Add connection = 1.0; Add node = 0.0; Mod Wgt = 0.9, Diff = 1.0; Avg. Weight = 0.5; Excess = 0.1}. With these coefficients, we ran NEAT for 3000 trials, no max generations, and a max score of 200. The resulting histogram can be found in Figure 16, where the average generation was 5.033 generations, which is a 50% reduction from our random sample.

An example neural network that we found works is showcased below in Figure 17. Each red line represents a positive weight whereas all blue lines represent a negative weight. The width of the line and the transparency represent the strength of the weight: the more thick the line, the larger the weight is and the more influence it has on the output nodes.

In conclusion, after creating everything from scratch, modifying, testing, and experimenting, we were able to create a neural network that can solve “Flappy Bird” in around 5 generations!

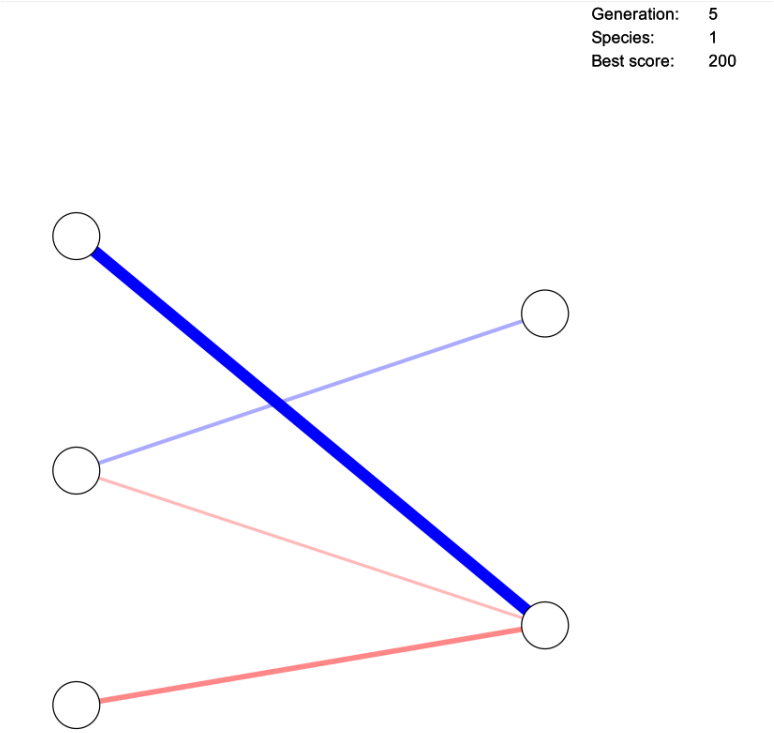

Figure 17: Example neural network created by NEAT.

Future Work

Given some more time, we would've liked to program the game of "PacMan" and "Snake" to create an environment to see how well NEAT performs on them. In addition to this, we would like to modularize our code for the NEAT algorithm so that it can be used as a template for future projects.

Conclusion

When playing "Flappy Bird", there are only 3 things a player needs to focus on: the bird's current y-location, the next incoming bottom pipe's y-location, and the next incoming upper

pipe's y-location. Using the key steps of the NEAT algorithm, we established an initial population, evaluated the population, speciated the individuals of the population, crossed over from within the species, and finally, applied a mutation on the offspring at chance. This process would then produce neural networks throughout each generation, eventually creating a neural network that could solve the game "Flappy Bird" indefinitely. To further enhance the performance of the NEAT algorithm, we used a modified hill climbing search to find the ideal coefficients for an instance of NEAT. This ideal instance of NEAT then produced a neural network in an average of 5.033 generations that was capable of solving "Flappy Bird".

References

Oken, A. (n.d.). An Introduction To and Applications of Neural Networks. 1–30. https://www.whitman.edu/Documents/Academics/Mathematics/2017/Oken.pdf

Oleg, G., & Alexander, N. (n.d.). Mutation Control in Neat for Customizable Neural Networks. 1–8. https://ceur-ws.org/Vol-2856/paper5.pdf

Qader, B. A., Jihad, K. H., & Baker, M. R. (2022). Evolving and training of Neural Network to Play DAMA Board Game Using NEAT Algorithm. *Informatica*, *46*(5), Article 5, 29–37. https://www.informatica.si/index.php/informatica/article/view/3897

Stanley, K. O., & Miikkulainen, R. (2002). Evolving Neural Networks through Augmenting Topologies. *Evolutionary Computation*, *10*(2), 99–127. https://doi.org/10.1162/106365602320169811

Wittkamp, M., Barone, L., & Hingston, P. (2008). Using NEAT for continuous adaptation and teamwork formation in Pacman. *2008 IEEE Symposium On Computational Intelligence and Games*, 234–242. https://doi.org/10.1109/CIG.2008.5035645