# CITS4404 Project Report

*Raph Byrne & Ash Tyndall*

## Introduction

For this project we decided to explore Genetic Programming and Neuro Evolution. These techniques were chosen for two different reasons.Firstly, they are both techniques that try to evolve some kind of controller. In the case of Genetic Programming we tried to evolve Decision Trees which control the robot through a system of interconnected rules  For Neuro Evolution we attempted to evolve fuzzy controllers through the use of a technique called Neuro-Evolution of Augmenting Topologies (NEAT) which uses evolutionary algorithms principals to evolve neural networks. The topology of the network is evolved during the process so this is like designing a fuzzy controller where we decide which antecedents affect each consequent. The weights between the neurons can be considered an abstraction of the definitions of the fuzzy sets in the fuzzy controller.

Secondly, both techniques utilise evolutionary algorithms to evolve real structures that control the virtual robots. After evolution these structures can be examined to determine the possible benefits of different control structures within robocode. For our Decision Trees this means we can look at the resulting tree and determine which actions and sensors are good choices for the objective being learning. For NEAT we can examine the topology of the resulting neural network and determine the learned function based on our defined inputs and outputs. We may also learn which inputs or outputs are more or less necessary based on the learned weights.

### Genetic Programming using Decision Trees

The branch of evolutionary algorithms primarily explored by Ash was the use of Genetic Programming techniques to generate decision trees that have the potential to inform strategies based on environmental parameters. The goal of implementation was to investigate movement and some minor targeting strategies could be used by the robot to allow it to win the game.

A simple tree-based "language" was designed that allowed the specification of both "Decision" and "Action" nodes that could be parents and children of each other in arbitrary arrangements. A Decision node would contain two child nodes, these nodes indicating the further nodes to "execute" given the Decision being evaluated as true or false, yes or no.

These trees formed the genotype of the system, and are generated randomly at the outset and evolved with mutation and crossover functions designed with trees in mind.

For the mutation function, a node is randomly selected from a candidate tree, and replaced with another node of the same type. This introduces variation into the population and helps prevent homogeneity from becoming the norm.

For the crossover function, a similar design is used; a node is randomly selected from each of two candidate trees, and these nodes, along with their subtrees, are effectively swapped. This is the primary operation by which genetic information is passed along to future generations.

Two different fitness functions were used when experimenting with this system; a simple truncation selection algorithm, which selects the best n/2 elements from a n sized list of the current generation and the previous generation, and a more complex roulette-wheel inspired algorithm, which ranks the same n elements, and assigns them each a probability of appearing in the next generation that causes a similar half reduction in elements.

## Neuro-Evolution of Augmenting Topologies (NEAT)

NEAT was developed by Stanley and Miikkulainen in [1] as an alternative to traditional neuro evolution techniques. The main advantage of NEAT is that while traditional neuro evolution fixes the topology of the neural network and only allows the weights to be evolved, NEAT allows for the topology of neural network to change over time. Also all networks start out as just a set of input and output neurons that are fully connected. These minimal networks are slowly mutated over time, adding new neurons and edges only when it is beneficial to the organism being evolved. This allows NEAT to develop very small and efficient networks to solve the requested problem.

NEAT skips the difficult procedure of performing topological comparisons through its genome representation. Each genome contains a list of nodes in the network and a list of weighted connections between those nodes. Each of the connections are tagged with an innovation number which allows for meaningful and fast reproduction. The innovation number is a globally incremented counter that updates whenever a new structural mutation occurs (a new connection is generated between two nodes).

Mutation in NEAT occurs in one of three ways. The first method is similar traditional neuro evolution, weights are either perturbed or replaced with some likelihood. The second method is the “add connection” method where a new weighted edge is added between node previously unconnected nodes. The third method is “add node” where an existing connection between two nodes is replaced by a new node that is connected between the two, adding two new weighted edges. Disabled connections have a small chance of being re-enabled during reproduction but are never removed as they might contain valuable genetic information.

Performing a structural reproduction of two large neural networks could be complicated but NEAT’s genome representation makes the process extremely simple. Each genome’s connection lists are sorted by their innovation numbers. Matching connections are inherited randomly from either parents. Disjoint connections (missing connections within the list) and excess connections (missing connections from the end of the list) are inherited from the fitter parent.

Speciation is very important in NEAT as without it NEAT has very serious diversity problems. Small networks are optimised much faster than larger ones so to not overly penalise large network there must be some protections. Speciation is done via a “topological distance” measure that estimates how different two genomes are. This is calculated as the weighted average of the number of excess connections, the number of disjoint connections and the average weight difference of the matching connections. If this distances is less than some defined threshold then two genomes are considered to be part of the same species. A representative of each species is selected at random to be used to generate the species of the next generation. Genomes within the same species share the fitness of the entire species. This protects exploratory genomes from being dominated by previous fitter solutions.

After speciation has been done selection can occur. Selection is a simple truncation selection algorithm where some proportion of each species is chosen to reproduce. Each species produces a number of offspring proportional to their share of the total fitness of the population, encourages search in the direction of successful species. The champion genome (with the highest fitness) of each species is taken to the next generation unaltered to prevent the loss of good solutions.

Some sections of the NEAT specification in [1] were not implemented due to time constraints. These include some small optimisations such as only allowing the top two species to reproduce if the average fitness of the population does not improve for twenty generations.

# Implementation

For our implementation, two separate projects were developed that explored different aspects of the Robocode environment and its potential as a fitness function for adaptive systems techniques. Both projects are written in Java, as it is the language the most stable Robocode bindings are available in, and they both utilise the RobocodeEngine API that is provided to interface with Robocode programmatically.

Frequent difficulties were encountered in the development process relating to Robocode's unique way of handling robot battles, and its incompatibility with adaptive learning techniques. With time and effort, methods using file I/O were developed to allow the genomes used in both techniques to be communicated to a generic robot class at runtime.

## Genetic Programming using Decision Trees

For the Genetic Programming project, a complex set of classes implementing abstract interfaces were used to generate a class-based representation of the "GATree", the core data structure utilised in this project. Each of these nodes had an "execute" action, that would firstly perform an action or a comparison of some sort, and then recursively execute its child node actions, depending on the specific implementation of that node.

To simplify the learning somewhat, the "GABot" class that executed these decision trees had some simple logic to help it avoid situations that required very complex logic to cope with. The two main things implemented were a constant targeting lock (the robot's gun would lock onto and track an enemy once it locates one) and a "bump" handler (if the robot collided with an enemy or wall, it would reverse a random amount and turn a random amount).

"Concrete" nodes (nodes implementing actual actions and decisions) were then created for tasks that would be useful in the examination of movement and basic targeting strategies. The primary "Concrete Action" nodes that were used include;

- Ahead - Moving the robot forward a given amount of pixels
- Fire - Firing a bullet of a given power
- Resume - Start moving
- Stop - Stop moving
- VaryGun - Vary the gun by a certain amount of degrees from its current position
- Turn - Rotate the robot body a certain amount of degrees

This "Concrete Action" nodes then had a random number generator associated with their constructors to generate values in appropriate ranges. These ranges were determined given consideration to the fact that the robot was likely to execute these actions once per turn, and they shouldn't be much larger than what a robot could physically accomplish in one turn (i.e. it shouldn't move ahead more than the maximum possible in one turn).

Similarly, "Concrete" nodes were then created for decisions that would be useful when decided on possible actions to take. These "Concrete Decision" nodes were based on both "just happened" events and also cumulative concepts, such as;

- Hit by bullets greater than, less than, this round
- Hit opponent greater than, less than, this round
- Energy greater than, less than
- Velocity greater than, less than

Much like the "Concrete Actions" , the "Concrete Decision" nodes use appropriately generated random numbers to determine the arbitrary values they compare against.

The system contained within GASystem handles the actual genetic programming operations of the problem. It provides functions for performing tree recombination and mutation, as well as initialising the entire system and evolving it over successive generations. It coordinates with the GATreeGenerator class to generate trees using reflection that can randomly contain any of the nodes specified in the config file. As well as specifying which nodes can exist, the config file also assigns these nodes a relative probability of them being used as a random node when constructing a tree. Nodes assigned a probability of 1.0 will always be chosen if the system selects them randomly, whereas nodes assigned a probability of 0.2 will only be chosen as the random node 20% of the time.

After this, there is the infrastructure surrounding the robot fitness calculation. By leveraging the robowiki.runner library, with modifications, the MatchPlayer class can accept the submission of any number of GATrees, and then queue them into a multiprocessing system that runs on 6 threads. This allows the fitness calculations to be performed in parallel such that results with appropriate numbers of generations can be achieved in reasonable times.

The MatchPlayer and modified robowiki.runner.BattleRunner class hook into the custom designed "FitnessResult" interface, which allows the specification of different types of fitnesses from the raw scores that the system collects. Right now only the score robocode assigns the robots is provided as a fitness measure, however it is easily possible to use other measures to allow for the optimisation of other factors.

Because of the evolutionary system's long running time before producing results, special care was taken to make it fault tolerant and recoverable from most errors. To that end, a timeout of (3 + 0.5 * tree size) seconds is imposed on the fitness calculation of each Robocode battle. Some genomes become corrupted and end up being passed through to Robocode, which hangs. In the case of this occurring, the Robocode process is terminated and reinitialized, and the culprit genome is removed from the population. To ensure consistent population size, random genomes take their place.

All of the aforementioned functionality mainly feeds into the EvolveOut program, which evolves a system with the given parameters, outputting fitness results and population snapshots to a directory.

To help the review of these files, the Playback program is also provided. This program can accept the path to a generation snapshot, run a more rigorous 15 round average fitness calculation on it and allow you to play a match with that genome at normal speed.

## Neuro-Evolution of Augmenting Topologies (NEAT)

NEAT was implemented almost exactly as specified in [1] minus the optimisations mentioned above. This generated a population of genomes that could be used to construct a neural network. Each genome was saved to a file and read in by a robocode bot which would generate a neural network and determine the fitness of using that network over a number of rounds. Each tick of a battle the robot would send the input of its relevant sensors to the neural network, wait for the output and then use that output to decide on the action to take during the next tick.

Input was collected each tick through the robot’s scanning and self status events. The input vector could be any length allowing the robot to react to range of environmental inputs. This was stored in a buffer and passed to the robot’s neural network during their run method. Once the neural network had produced outputs they were fed back into the robot’s actuators such as movement, turning and firing. The outputs were scaled up to match that control’s range.

There were no predefined behaviour in the robot that the neural network could select from. Any behaviour the robot exhibited had to be a function of the inputs to the outputs. This somewhat limited the learning capability of the robot as all of its actions had to be decided every round. Any learning of a sequence of actions (which can be quite important to robocode) had to occur by evolving recurrent structures in the NEAT networks. Ultimately the robot was only able to learn recommended good turn by turn movements given the environment with little memory.

# Experiment Design

## Genetic Programming using Decision Trees

To test the effectiveness of the GP techniques used in this project, several experiments were designed to test how the varying of parameters affected the learning rate of the system, and the ultimate fitness. The parameters that were experimented with include;

- Tree depth
- Tree complexity
- Number of rounds when calculating score
- Number of generations
- Number of genomes

To gain an idea of how those parameters come into play, five different experiments were conducted. For these experiments, the following parameters remained constant and were not experimented with, predominantly due to time constraints;

- Proportion of decision nodes to action nodes; 1:8 ratio
- Mutation rate; 5% chance per tree
- Battlefield size; default 800x600
- Gun cooling rate; default 0.1
- Inactivity time; default 450
- Robot tested against; sample.Walls
- Fitness measure; truncation selection

The fitness measure used was the difference in score between the opponent robot and the robot being tested. As such, negative fitness represents losing, and by what degree, and positive fitness represents winning, and by what degree.

For these experiments, truncation selection was used in all cases, except when noted.

## Neuro-Evolution of Augmenting Topologies (NEAT)

For this project we mainly focused on a robot that could aim at a stationary target. The inputs to the evolved neural network were:

- The robot's gun heading
- The robot's body heading
- The bearing to the enemy

The outputs were:

- The amount in radians to rotate the gun this round

The scanning procedure was fixed to create constant scans of the enemy every tick and both robots were stationary. This minimised the possible noise of the system. Also inputs and outputs were scaled down and up respectively. We scaled the inputs down into the range [0,1] to try and speed up the learning rate and we scaled the outputs up so that they became sensible control inputs into the system.

Effectively we were trying to learn the function:

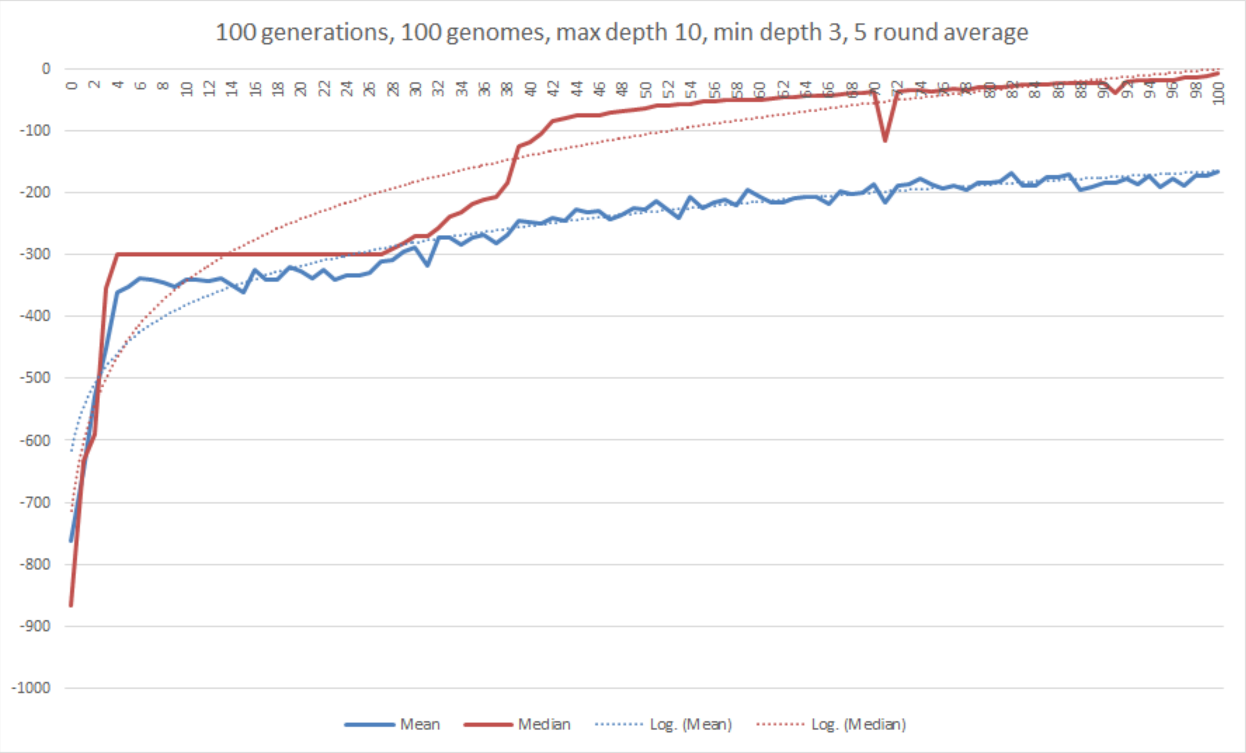
gun turn amount = our heading + enemy bearing - our gun heading

As this is a linear function it should be possible to create a neural network that can perform this at most one hidden neuron however noise in robocode can make this difficult. This objective was chosen as it is a very simple learning objective.

The gun for our robot was set to fire periodically at a constant power and the fitness function was set to the percentage of hits the robot achieved out of the total number of bullets fired over five rounds.

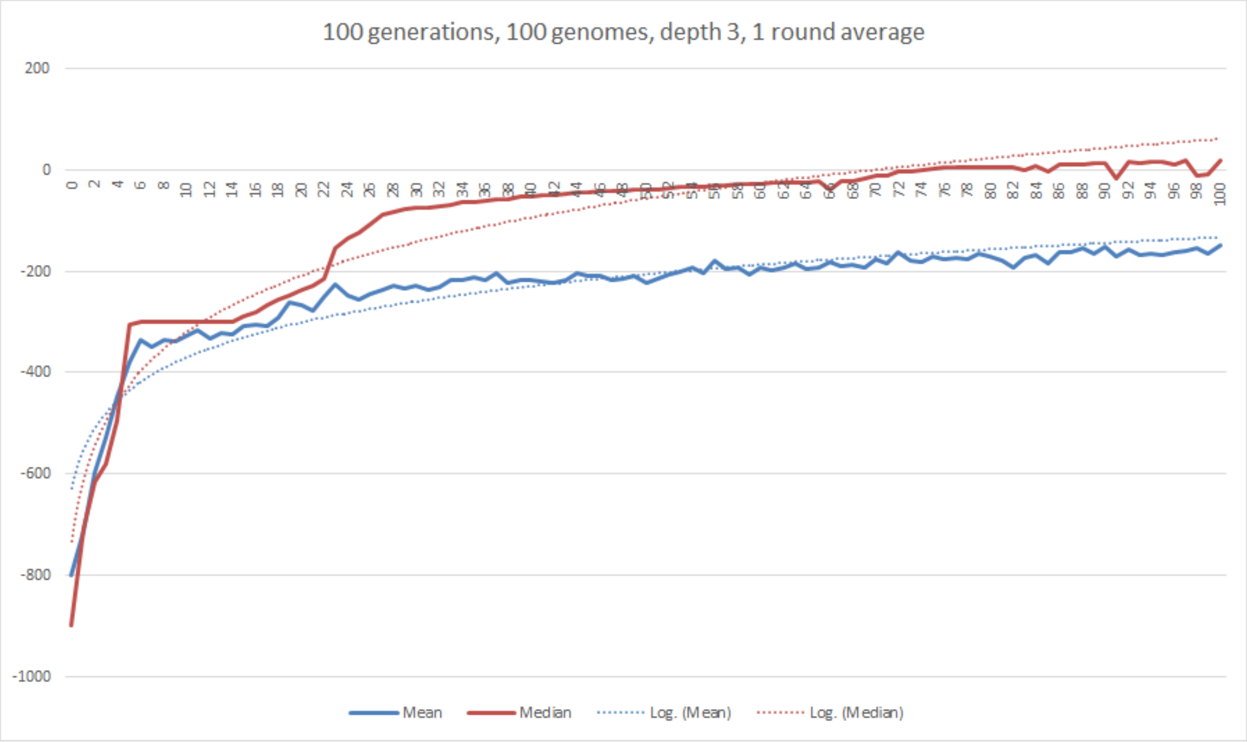
# Results and Conclusion

## Genetic Programming using Decision Trees



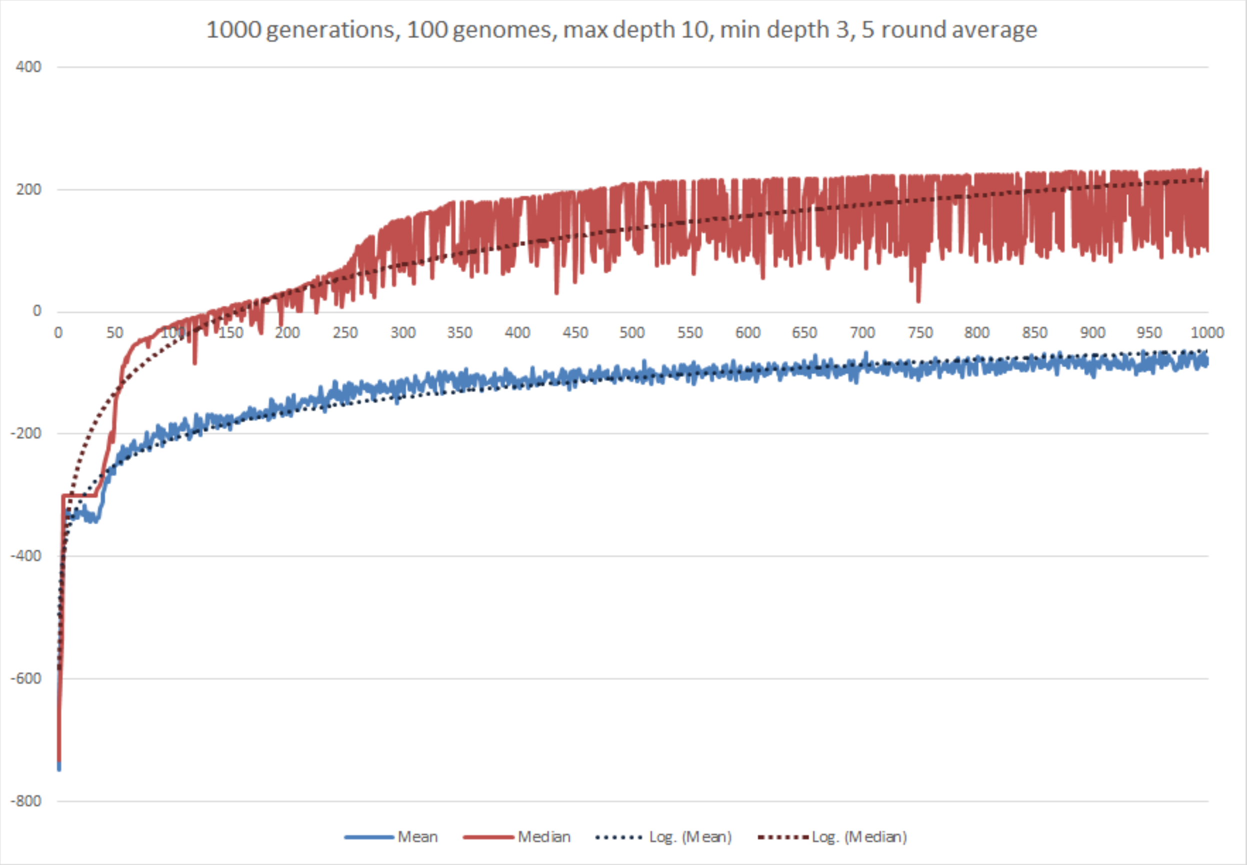
This experiment can be considered the baseline by which all other experiments can be compared. We can see from the results it shows that the learning rate approximates logarithmic: The system doesn't much until around the 40th generation, after which it has successfully recombined genomes that begin to demonstrate the characteristics of fitness improvement.

However, we can see from these results (which are representative of the whole generally) that the fitness unfortunately tapers off after this initial spike, and does not succeed in producing a decision tree that is superior to the strategy employed by the sample.Walls bot.



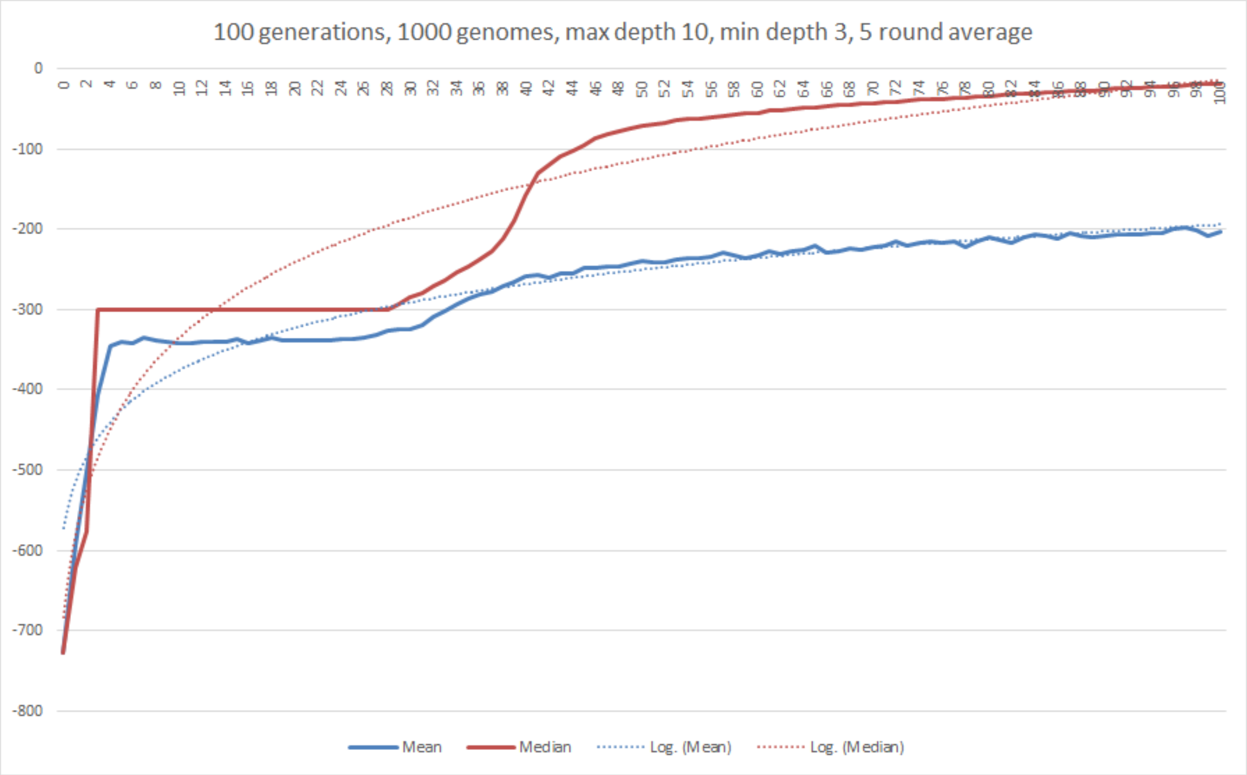
These results are interesting when compared approximately the same parameters, but with a more shallow depth and no multi-round averaging. This simulation shows the same sudden increase in fitness as the system starts finding better strategies, and notably ends up in the positive fitness in some instances.

This can be attributed to the fact that without averaging, the random placement of the different bots can lead to more favourable results in which the tested bot wins purely by chance.



To see a more long-term view of the fitness rate, a setup was run with 10x as many generations as the control sample. As you can see, the fitness clearly follows a logarithmic progression; increasing rapidly, but then slowing down.

This data however seems to suggest that the average fitness, while certainly slowing, would eventually break into the positive fitness measures given enough time. However it is difficult to see if the fitness truly plateaus with so little data.



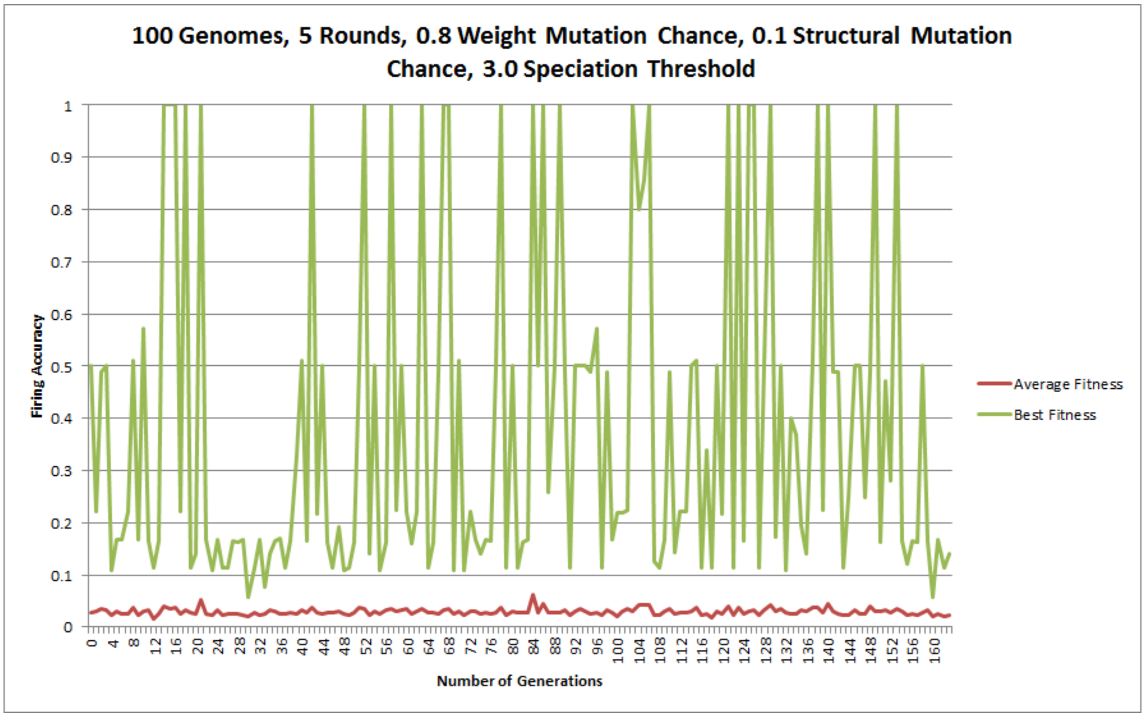
Increasing the number of genomes in the system by 10-fold had the effect of smoothing out the fluctuations in the system and producing much cleaner data, however it did not change the learning rate or ultimate fitness by any appreciable amount.

In conclusion, we can see that a decision tree is a difficult medium to evolve a viable competitive strategy for Robocode. Between the fact that fitness takes long periods to calculate, and that decision trees do not encourage much reactivity or complex strategies, most strategies that were evolved consisted of some simple repetitive movement (such as rotating continuously) and the Fire action. These strategies appear to be the most simple decision trees possible that produce some results, and thus were selected by the system.

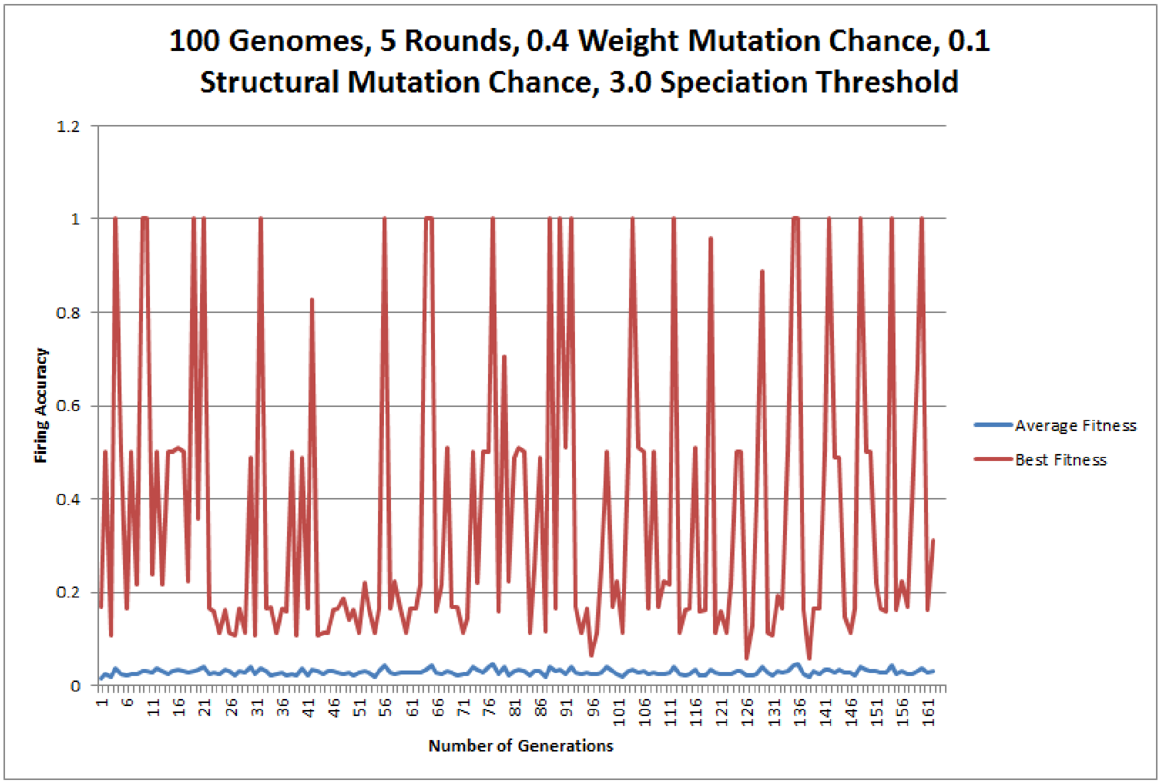
## Neuro-Evolution of Augmenting Topologies (NEAT)

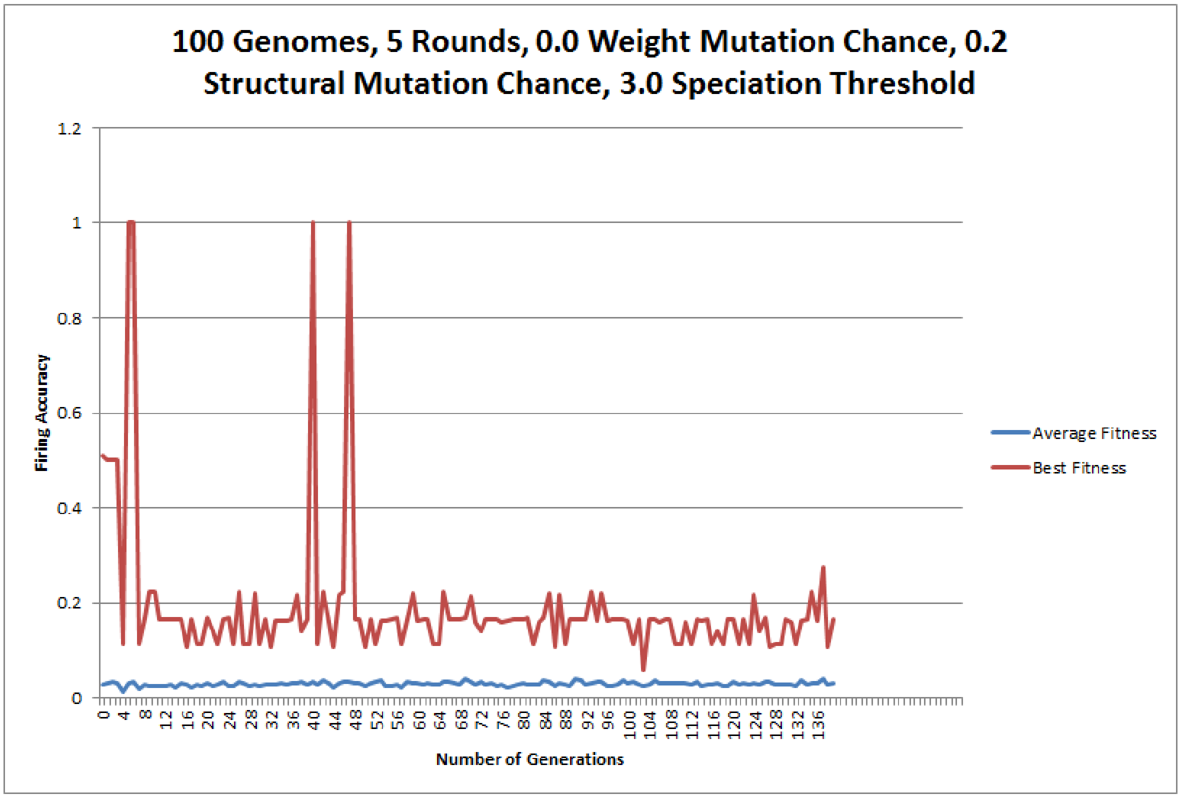
There are a number of different parameters to NEAT and robocode that we varied across our experiments. These include:

- Population Size
- Number of robocode rounds to train for
- Probability of mutating a weight
- Probability of performing a structural mutation
- The maximum distance between two genomes in the same species
    - Lower numbers indicate more, small species



This graph represents running a test using the recommended parameters from [1]





We can see that even with different parameters that NEAT performed poorly at learning the required aiming function. Although occasionally good solutions could be found within the population it seems that either our NEAT implementation was unsuccessful in using their genetic information to improve the overall fitness of other genomes or the environment was too noisy (with only five rounds) for accurate fitness measurements to be made.

Whilst it remains possible that our implementation of NEAT was inadequate it may also be the case that NEAT is not applicable to solving the targeting problem. Given more time it would be interesting to explore different fitness measures or control schemes. In particular it may be the case that in robocode it is more useful to use the neural network as a soft action selector rather than a direct controller. Also as the function we are trying to learn only requires very small neural networks NEAT may not be the best tool to use as the process of NEAT to try and create larger networks when it believes that smaller networks cannot solve the problem, which may happen erroneously due to the noise in the robocode environment.

In conclusion it seems to be that case that due to environment noise and the way that NEAT tends to grow its networks quite rapidly that NEAT is not useful in solving the targeting problem in robocode. NEAT may be more useful in solving more complex robocode control problems were larger networks are required or as a soft action selector for pre-built robot strategies.

# References

[1] Stanley, Kenneth O., and Risto Miikkulainen. "Evolving neural networks through augmenting topologies." *Evolutionary computation* 10.2 (2002): 99-127.