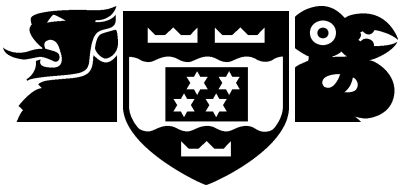
VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui



Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@mcs.vuw.ac.nz

Dynamic Neural Architectures

Justin Rens

Supervisor: Marcus Frean

October 2003

Submitted in partial fulfilment of the requirements for
Bachelor of Science with Honours in Computer Science.

Abstract

Evolving controllers for robotics is a fantastic proposition. Evolution often finds simple, yet efficient controllers for solving problems and this is a clear advantage for situated robotics. Evolution of neural networks can be broken into direct input-output mappings, evolved network structures with off-the-shelf learning algorithms, and statically sized dynamic neural networks. This paper focuses on evolving dynamically sized neural networks with dynamic update rules. It is shown that by doing this networks can be evolved to solve the task without having to specify either the size and connectivity of the network, or the type of learning rule to apply to the network.

Acknowledgments

The people who have provided much needed help to the project:

- For all the help and advice in both getting through the ideas, and editing the report, and for allowing me to explore the topic openly I thank my supervisor Marcus Frean.
- For help in helping me get my head around some difficult ideas, and for generally bouncing ideas I would like to thank Jeromé Dolman.
- For editing my report and providing some ideas outside of the field of computer science I thank my parents.
- To the other people who have contributed to either my research, or keeping my sanity: Bunna, Annie, Will, Simon.
- Finally to myself, without whom this report would not have an author.

Contents

Figures

Chapter 1

Introduction

1.1 Background

The use of evolutionary techniques to build controllers for robotics is a tantalising proposition. Evolved controllers have been shown to be remarkably good at finding simple, yet efficient methods, to perform tasks by taking advantage of quirks or geometric patterns in the environment in which they were trained [9]. These controllers have been able to find means of navigating mazes with similar behaviour to rats without using complex environmental mappings but by using reactive sensory-action sequences that exploit the general layout of the environments. The ability of these controllers to solve tasks without complex representations of the state of the world (or themselves) gives these controllers quick and efficient real-time operation. This gives them a clear advantage for use in real world autonomous robots. However these controllers tend to lack adaptability to changes in their environment, sensor changes, or even migration to other robotic bodies.

Methods have been suggested to improve these robots by adding noise to the sensory input taken from the environmental, or training the robots in several differing environments (Jakobi 1997 [7]). These methods though, suggest that you know what the change in the environment is going to be in advance. An alternative method is to employ learning over the lifespan of the robot.

A philosopher Daniel Dennett proposed that there are three types of creatures that can be evolved.

1. The first type is the traditional *Darwinian creature* that is evolved to create a direct mapping between its sensors and its actuators. These are the simplest creatures, and while they may be evolved to work well in a particular environment, they are very fragile when moved between environments.

2. The second type is the *Skinnerian creature*. This creature is capable of learning the behaviours that should be applied under different circumstances, but is unable to make predictions about the world.

3. The third type of creature is called a *Popperian creature*. This creature is capable of building internal models of its world, and can test its selection of behaviours against this internal model. This allows it to learn without exposing itself to potentially disastrous situations (i.e. stepping in front of a bus to see if it hurts).

This pyramid of evolved creatures provides a roadmap that can be used to advance from the basic robotic controllers to more advanced planners. The first step would be evolving the direct mapping from inputs to outputs. This is the most common approach used to

evolve controllers and generally involves evolving the parameters for a neural net. However due to the nature of evolution these controllers are often very good at finding tricks that allow them to take advantage of their initial environment, but fail when moved to different environments.

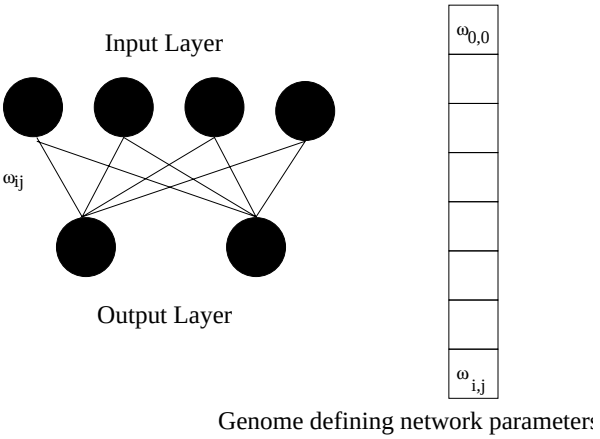


Figure 1.1: A Darwinian creature with the network parameters explicitly evolved in the genome.

Proceeding to the next tier gives controllers the ability to learn the structure of their environment. To do so a neural net is given some means of learning, such as Direct Reinforcement or Back Propagation, and the number of nodes, and inter-connectiveness of the network is determined by using evolution. This method is far more flexible than the Darwinian creatures as it can do online learning during the lifespan of the creature, but is still limited by the learning algorithm that the designer chooses. This makes the general assumption that these learning methods are in fact the best suited for the task.

1.2 A Dynamic View of Control

Much research into situated controls thus far has focused on finding a set of parameters that will cause a network to function. For instance, a neural network trained with back propagation is updated until the weights (the parameters) of the network are found. At that point the neural network is considered fully trained and its training is turned off. If we use the analogy of a mountaineer climbing a hill, then a parameterised model uses a generic rule to find a specific point; in this case, move towards the ground that is the steepest from you until you find the highest bit of ground.

Recently new research has been focused on finding dynamical ways to update neural networks. Dynamic systems focus on finding a set of rules (rather than parameters) that define how to update the set of parameters. If we go back to our mountaineer example, we say that a dynamic system will try to find how to move so as to find the highest point. A dynamic system therefore learns to learn.

The layout of this 'ladder of learning' can be described as:

- the lower level being the set of parameters, or the point in search space that you're looking for,
- the second level being the rules by which you modify your parameters,
- and the third level as the rules by which you learn to modify yours rules.

Floreano et al. proposed that rather than simply evolving the structure of a neural network then applying some learning rule, the learning rule could itself be evolved as part of the structure of the neural network [10]. He argued that by encoding the update rules into a network rather than the weights, the networks would be forced to evolve on-the-fly learning methods. Also, the evolutionary pressure of adaption (slower adapting networks would suffer reduced fitness) would generate fast-adapting controllers. His work involved using neurons with differing localised learning rules and using evolution to choose which type of neurons should be used in the network.

The types of neurons that were used were based on the research of Chalmers (1990 [3]) who showed that combinations of these neurons could replicate the Back Propagation algorithm with only localised information. Also they were based on neuropsychological studies that showed that these were the most common mechanisms found in mammalian nervous systems (1990 [11]. They are:

- **HebbNode:** A typical Hebbian (D.O. Hebb 1949 [6]) node that increases the strength of the synapse proportionally to the activity of the pre-synaptic node x_j and the post-synaptic node y_i.
$$\Delta w_{ij} = (1 - w_{ij})x_j y_i \tag{1.1}$$

- **PostNode:** This node behaves in much the same manner as the HebbNode but will weaken the synapse when the post-synaptic node is active and the pre-synaptic node is not.
$$\Delta w_{ij} = w_{ij}(-1 + x_j)y_i + (1 - w_{ij})x_j y_i \tag{1.2}$$

- **PreNode:** Same as the PostNode only this one weakens the synapse when the pre-synaptic node is active and the post-synaptic node is not.
$$\Delta w_{ij} = w_{ij}(-1 + y_i) + (1 - w_{ij})x_j y_i \tag{1.3}$$

- **CovarianceNode:** This node makes its synapses stronger when the two nodes have similar activity levels otherwise it makes them weaker. So the synapse is strengthened if the difference between the two nodes is less than half their maximum output.
$$\Delta w_{ij} = \begin{cases} (1 - w_{ij})\mathcal{F}(x_j, y_i) & if \mathcal{F}(x_j, y_i) > 0 \\ (w_{ij})\mathcal{F}(x_j, y_i) & otherwise \end{cases} \tag{1.4}$$
where $\mathcal{F}(x_j, y_i) = tanh(4(1 - |x_j - y_i|) - 2)$ is a measure of the difference between the node activities. [1]

In Floreano's paper he did a comparison of using the dynamic adaptive methods versus standard evolved input-output mappings (Darwinian creatures). In his paper the robots had to move to a light switch, turn it on, then travel back towards the light to score points. He showed that the robots that had controllers with dynamically encoded learning rules were evolved faster, and performed better than the weight encoded models.

Floreano's method used a fully connected network of 12 nodes. Each node (or synaptic connection as desired) was encoded on a string of bits that determined the type of node. A genetic algorithm was applied to the string to evolve the network until it was capable of performing the task. Whilst Floreano's work proved to be a success it appeared that his technique still suffered from the same drawback as the traditional methods in that it assumed that the problem could be solved by only 12 nodes.

[1] In these formulae the $(1 - w_{ij})$ is used to limit the values the weights can take to a value between 0 and 1.

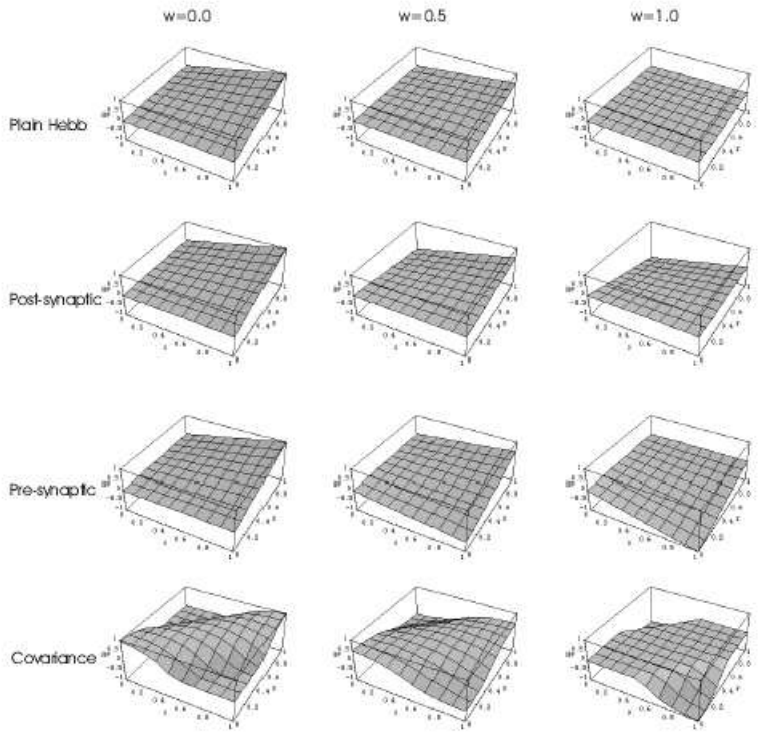


Figure 1.2: The learning behaviour of the different nodes.

1.3 Dynamics Neural Architectures

The topological structure of the search space involved in the creation of neural network controllers consists of many locally optimal peaks. Or in other words, when searching for a functional neural network, many controllers will be found that can solve parts of the problem, but may not necessarily be correct for solving the entire problem. Conrad proposed that hyper-dimensional bridges may link these peaks together and thus allow a learning controller to easily move from one peak to another [4]. However by strictly specifying some aspect of the neural networks design (such as learning rules, network size, network structure, etc) these bridges can be removed from the search space.

This idea of applying restrictions to a search algorithms space can be detrimental. An example of this effect can be demonstrated by observing a network designed to solve the XOR problem. Looking at figure 1.3 we can see that by restricting the size the number of nodes we reduce the set of problems it can solve to either logical AND or logical OR. However by adding an extra node the network becomes capable of performing logical XOR.

This idea scales to larger domains and in these, the application of restrictions to the network size, learning rules, connectiveness, etc, can have one of three effects. First, the restrictions applied serve to narrow the search space, and improve the likelihood of finding an adequate controller. The second effect is that the restrictions remove the hyper-dimensional bridges linking possible solutions, increasing the chance that controllers will fail to proceed past local optwere imum. The third and most detrimental of these effects is that the space no longer contains any solutions and no controllers will be found that are effective in the domain.

To do so we look at a method of letting the evolution cycle find controllers without imposing any restrictions (such as learning rules, network structure etc). The solution pre-

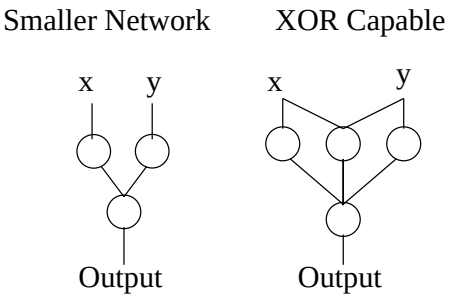


Figure 1.3: The effect of restricting aspects of a neural network can adversely affect its capabilities.

sented in this report by Dynamic Neural Architectures (DNA) is to allow the evolutionary process to freely find both the structure and the learning rules that the network should employ. To do so it hybridises Floreano's evolved learning strategy with the dynamical structures involved with the more traditional methods.

This report describes the developement of Floreano's work to extend his concepts to use a richer variety of network structures. It then does an experimental investigation into the results of the algorithm describing its abilities and restrictions. were

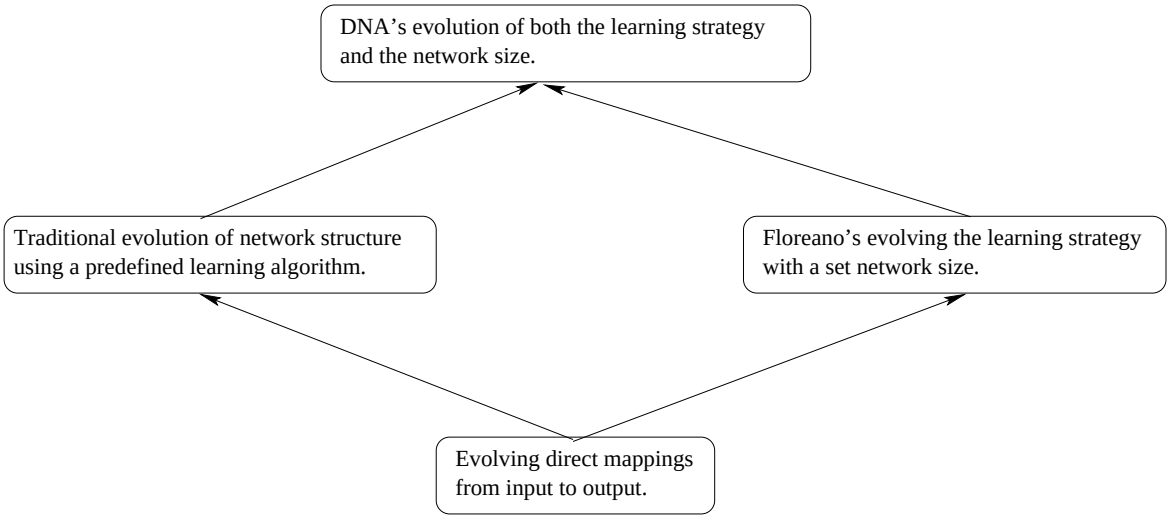


Figure 1.4: Where DNA fits in relation to other neural evolutionary techniques.

Chapter 2

Building Dynamic Neural Architectures

The chapter focuses on the development of the components of DNA. It looks at how the networks were built, how they functioned, the methods that were used to find the structures of the networks, and the construction of the world. It also looks at how the neural networks were trimmed to their functional core, and then analysed.

2.1 The structure of the brain

The brain[1] is modelled as a directed cyclic graph with a delay propagation scheme across the nodes. The graph is separated into an input layer, processing layer, and the output layer. Each layer has restrictions to the types of nodes that can exist in that layer as well as the types of connections allowed.

- In the input layer, nodes may have outgoing connections but may not have incoming connections. Their learning weight is simply $\Delta w_{ij} = 0$. The size of this layer is fixed as it is determined by the number of sensors on the robot.
- The processing layer can contain Hebbian Nodes, Pre-Synaptic Nodes, Post-Synaptic Nodes, or Covariance Nodes. These nodes may have both incoming and outgoing connections. The learning rules for each of these nodes are described in Chapter 1. These learning rules are applied to each of the nodes outgoing connections (i.e. connections from the current node to the post-synaptic node). The size of this layer is unspecified and may be anywhere upwards of 0 nodes.
- The output layer nodes have no outgoing connections. There are no specifications as to how many incoming connections the output nodes may have, and so they may be connected from one node, many nodes, or even have no incoming connections. They have no learning rule as they have no connections to apply the rules to. The size of this layer is fixed to the number of actuators the robot has.

Each node within the network is derived from a basic sigmoid function node. At each brain cycle the node sums its inputs, applies the sigmoid function to this value to get the nodes activation level, and then propagates this to the next level of nodes (who themselves receive the input at the next cycle). Then the node updates its weights according to its own

[1]The use of the word brain is used loosly to mean controller or neural network. It is by no means an reference to an actual biological brain.

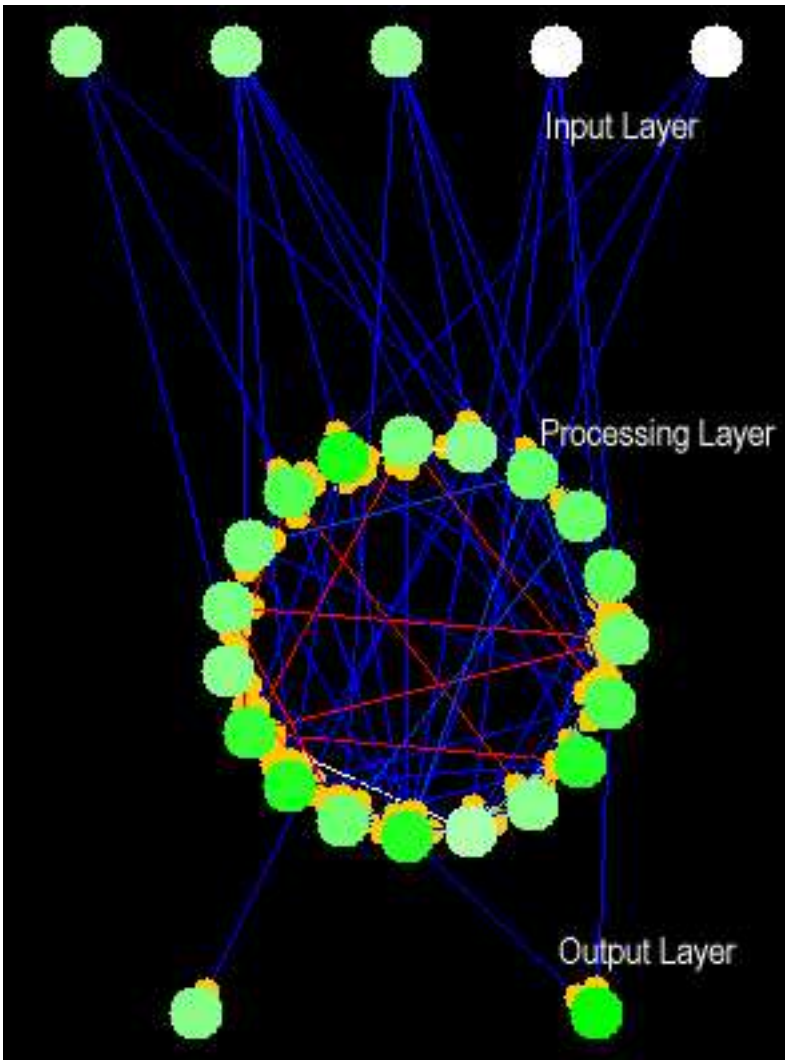


Figure 2.1: An example of the structure allowed. The graphical layout of the controller is arbitrary and is laid out in a circle only to make displaying easier. As a key, the more green a node (circle) is, the higher its output activity. The weights of the connections are coloured from red being the most negative, to white being neutral, to blue being the most positive. The yellow blobs represent the synaptic connection and are used to show to which node a synapse connects.

learning rules and uses the post nodes previous output (i.e. the output at cycle t-1). The exceptions to this rule are the output nodes who simply have a linear output function (their output matches whatever their input is), and the input nodes who do not use a learning rule.

2.1.1 Characteristics of the Network

Giving each node a propagation delay allows the network to put together structures that can utilise a time component. Several effects that may not be possible with a single update network are shown here:

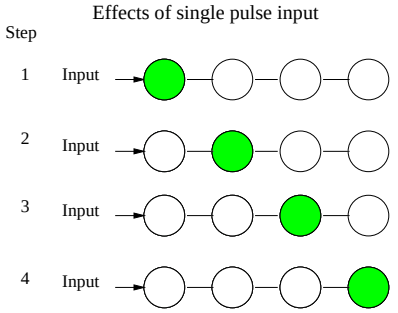


Figure 2.2: An example of a single pulse being propagated through the nodes.

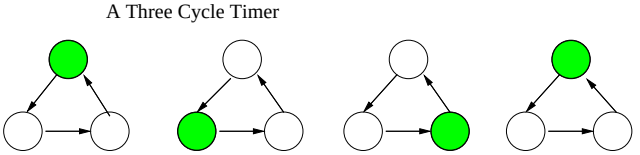


Figure 2.3: An arrangement of nodes that will build a three cycle timer.

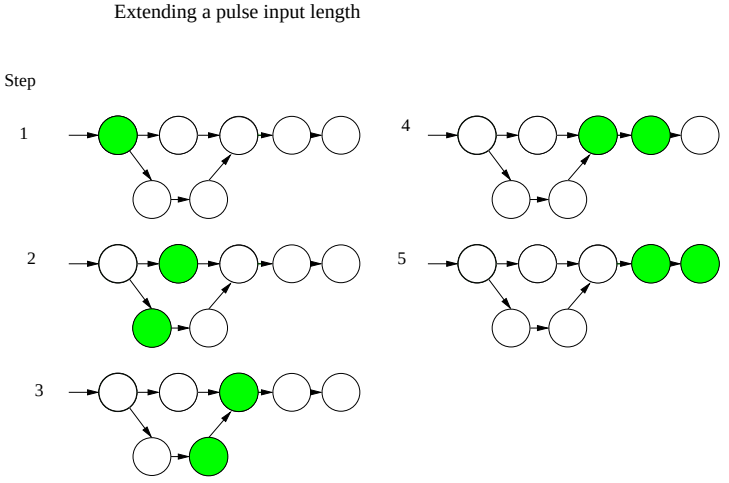


Figure 2.4: Extending a single pulse input to a dual pulse.

2.2 The Evolution of Network Structures

Given the structuring of the brain, some means of specifying the design that could be used by a genetic algorithm was required. In order to implement the algorithm two functions

need to be considered, namely:

- Cross-over: Takes the genomes of two individuals and splices them together to create a third individual.
- Mutation: Randomly performs some point change to the genome of a single individual.

There are two approaches to implementing a genetic algorithm. In the first approach the genome is represented as a string. Here, cross-over involves swapping substrings, and mutations involve changing some bit along the string. The second approach uses a tree to represent the individual. In this case cross-over is performed by swapping subtrees and mutation involves changing some subtree.

2.2.1 Evolution: Version 1

My early experiments represented each individual as a string of data. Figure 2.5 shows how a gene was encoded. A gene sequence is terminated when the beginning of the next gene is found. The relative connections were based on the number of nodes in the processing node ahead of the current node with wrapping. For example, in a five node controller, if node 1 had a pointer connecting to node 7 that would actually be a connection to node 2. This is shown in figure 2.6.

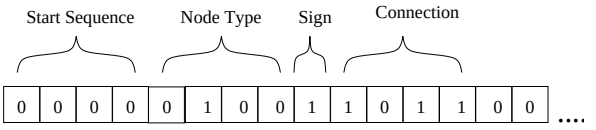


Figure 2.5: The initial string based encoding of an individuals controller. Here the amount of space reserved for a connection is 4 bits. This means a node can connect to a maximum of 15 nodes from its current position. The number of connections repeats until 4 consecutive zeros are read.

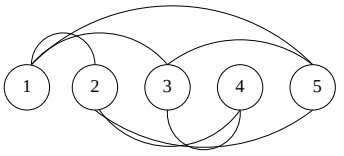


Figure 2.6: Translating the genome to a controller.

Node	Relative Connection
1	1,2,4
2	None
3	2
4	3,4
5	2

This version of encoding the individual offers a number of advantages. Firstly, because it uses a string based genome, it is very easy to apply an 'off the shelf' genetic algorithm to it. Also, translating the genome to the controller could be performed easily by building the controller as the string is parsed.

However a significant drawback is that any modification to the overall structure of the genome drastically changes the network's layout. For instance, if node 3 in figure 2.6 was removed, node 1 would have a connection to itself. This causes a substantial change from the original structure even though only a single node was removed. This effect made searching for functional structures too difficult.

2.2.2 Evolution: Version 2

The second version of encoding the controller into some form of genome uses a tree structure. Encoding the graph structure into a tree is essentially the same as 'unrolling' the graph. This is done by wrapping the graph nodes into TreeNodes. A TreeNode held one graph node, but it can hold the same graph node as some other TreeNode. This allows cyclic connections to be encoded into the tree.

Within the genome, multiple trees are allowed that have as their roots each of the inputs and are terminated either by having an output node or by reaching some predefined depth. Figures 2.7 and 2.8 show how the tree structure used in the genome relates to the final graph structure. The numbers represent the graph node that is being stored in each TreeNode, and can also be seen in the graph.

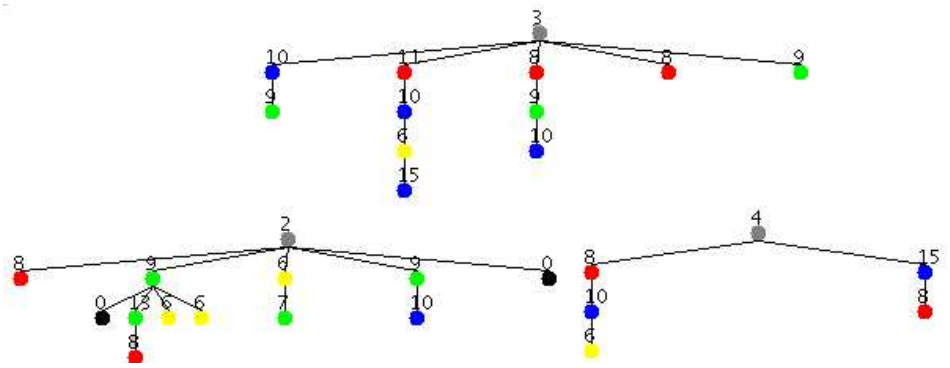


Figure 2.7: The tree structure that the evolution algorithm uses. Each tree grows from a single input. The tree is built up from the root outwards until either an output node is added, a node connects to all the other nodes, or a maximum depth is reached. Note each node in this graph is a TreeNode and wraps the graph nodes, hence the numbers being repeated in each tree.

This method proved once again to be very simple to build an initial individual, and though in theory it would be easy to pass the tree structure to a standard genetic programming package, in practice keeping track of which node went where became too difficult.

2.2.3 Evolution: Version 3

After looking at the previous versions a new strategy was required to evolve the necessary graph structures. This new approach involved throwing out the idea of an evolved structure that could be mapped to a graph structure, and instead focused entirely on using the same structure to both evolve the individual, and to function as the controller. Previous work into using graphs as the genome in an evolutionary process was relatively sparse, but a team from NASA (John Lawton and Todd Wipke [8]) had done some work on using genetic algorithms to search for chemical molecules. Although their application domain was very different, these molecules can be described as a graph, and so their work was quite applicable.

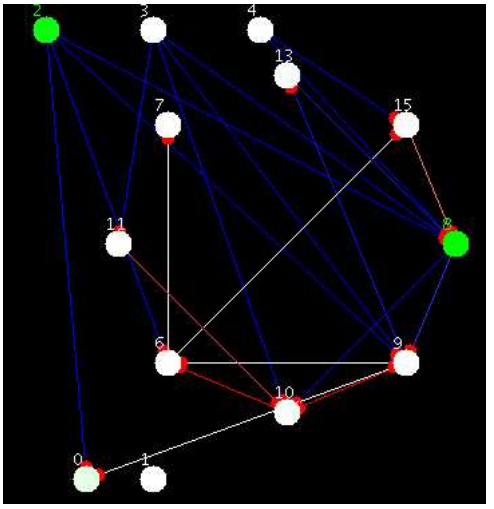


Figure 2.8: The tree structure of figure 2.7 translated into a working controller.

They focused on breaking and reforming bonds between atoms in a molecule. During a cross-over function they made sure that the number of bonds broken was almost as equal to the number of bonds formed. This is equivalent to saying that they tried to preserve the number of edges broken between vertices in a graph and the number of edges formed.

Their approach was as follows:

> ” To divide a molecule into two fragments we use the following procedure:
>
> 1. Choose an initial random bond
> 2. Repeat
>   (a) Find the shortest path between the initial bond’s vertices (the first time this will simply be the initial bond).
>   (b) Remove and remember a random bond from this path. These bonds are called ”broken edges.”
> 3. Until a cut set is found, i.e., no path exists between the initial bond’s vertices.
>
> To combine fragments we use the following procedure:
>
> 1. Repeat
>   (a) Select a random broken edge. Determine which fragment it is associated with.
>   (b) If at least one broken edge in [an]other fragment exists
>   i. choose one at random
>   ii. merge the broken edges into one bond; respecting valence by reducing the order of the bond if necessary
>   (c) Else flip coin (this step was disabled by a bug in (Globus [1])
>   i. if heads – attach the broken edge to a random atom in other fragment (respecting valence)
>   ii. if tails – discard the broken edge
> 2. Until each broken edge has been processed exactly once”

This process is summarised in figure 2.9.

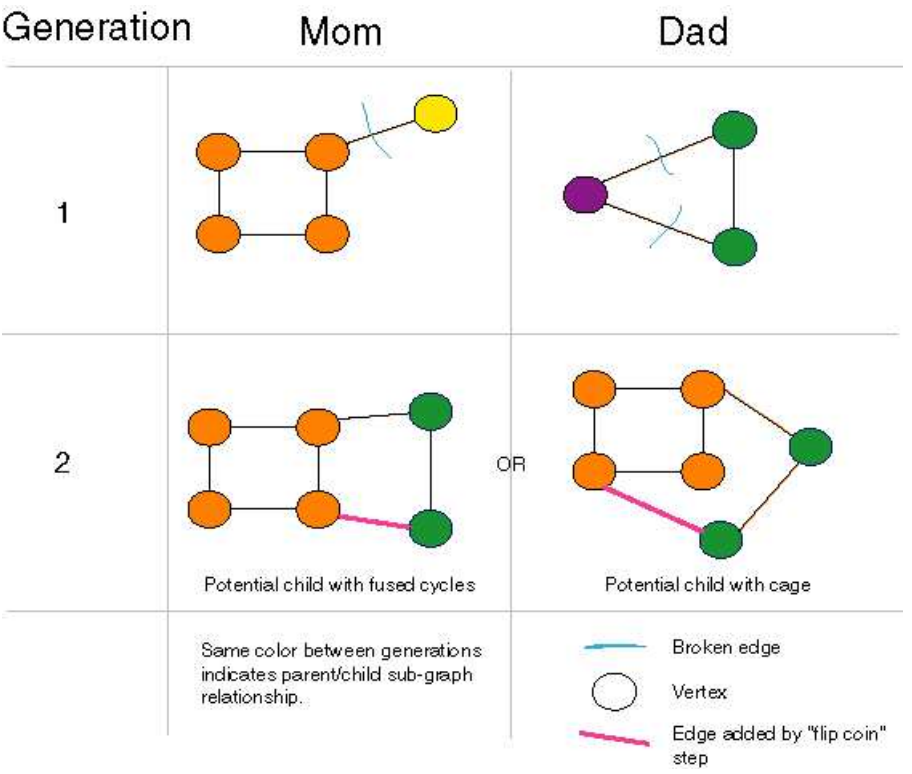


Figure 2.9: How the NASA Team did Graph Evolution.

For DNA however this process was modified to a simpler form by performing a multi-subgraph cross over. The first aspect is that this method only produces one child (which is slightly different from the traditional method that produces two children). This is due to the subgraph of the foster parent replacing a subgraph of the main parent. The recipe for this procedure is:

1. Select an equal number of random nodes from each of the parents to produce lists of equal size.
2. For each node being replaced;
   (a) Copy all the incoming and outgoing connections from the initial node to the replacement node except for nodes that are members of the replacement list.
   (b) Preserve the connections between the nodes that are members of the same list.

The advantage of this method over any of the previous methods is that only a small number of changes are made to a graph during cross over. During the breeding cycle the only changes made to the genome of the individual are the changes to the node types of the nodes that are replaced, and the internal connections between those nodes. Any other nodes and their connections remain unchanged as well as both the inbound and outbound connections to the members of the replaced nodes. This is demonstrated in figure 2.10 where despite the difference between the two genomes, only some small changes (shown in green) are made to the graph.

This method was chosen after testing to show that the changes made to a graph are only to the subgraphs being replaced. This was done by breeding a single individual with itself. This would keep the same underlying structure (except for mutation), but change the types of nodes and connection weights. This is shown in figure 2.11

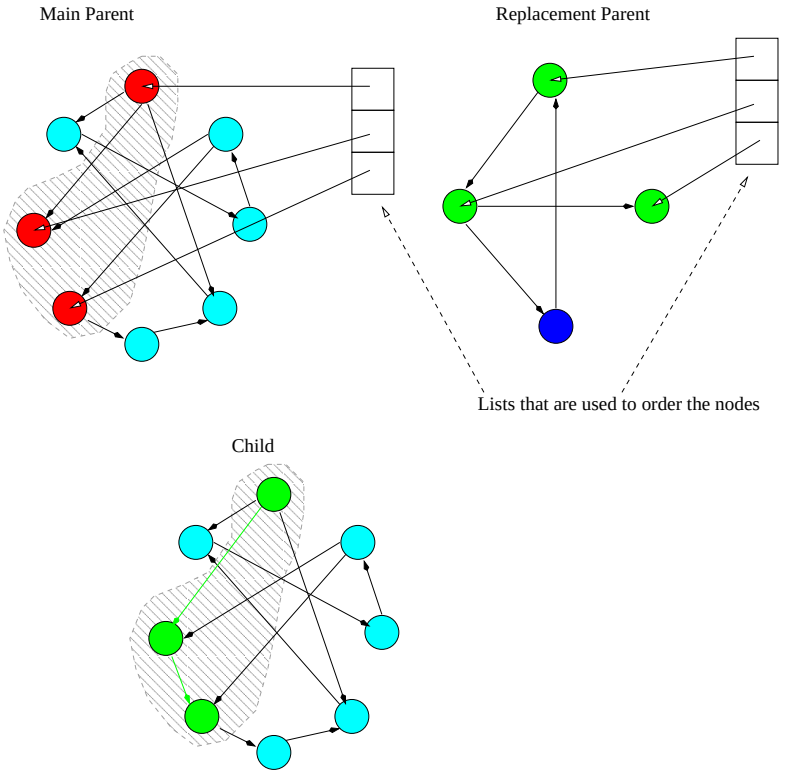


Figure 2.10: Subgraph replacement to create a new child

2.3 The World

2.3.1 What Inspired the World

In order to test the DNA mechanism, a domain was needed that would be initially simple, but could be extended for more exhaustive testing. The domain would also need to provide some indication of partially correct solutions so as to provide the evolutionary aspect of the project with a smooth fitness gradient to grade the individuals. Finally, it needed to have some real world component that would give some means of checking the validity of the simulation.

A good domain that met these requirements was line tracking using a simulation that modelled the Lego Mindstorms system. Line tracking is the ability to follow a trail or marking along a surface. In nature this particular task is almost an essential behaviour of any mobile creature ranging from ants following scent trails, birds navigating via polarised lines, or fish following ocean currents. Its also useful for driving vehicles on roads. This domain suited the requirements well because:

- Partially correct solutions, such as following the line for some distance, can be rewarded.
- The domain could be extended from an initially simple task of following a straight noise free line, to curved lines, intersecting lines, and even dislocated lines.
- By using the Lego Mindstorm system it is possible to model the simulation such that it can accuratly account for noise, the dynamics of the robot movement, as well as providing a platform to test some handwritten rules.

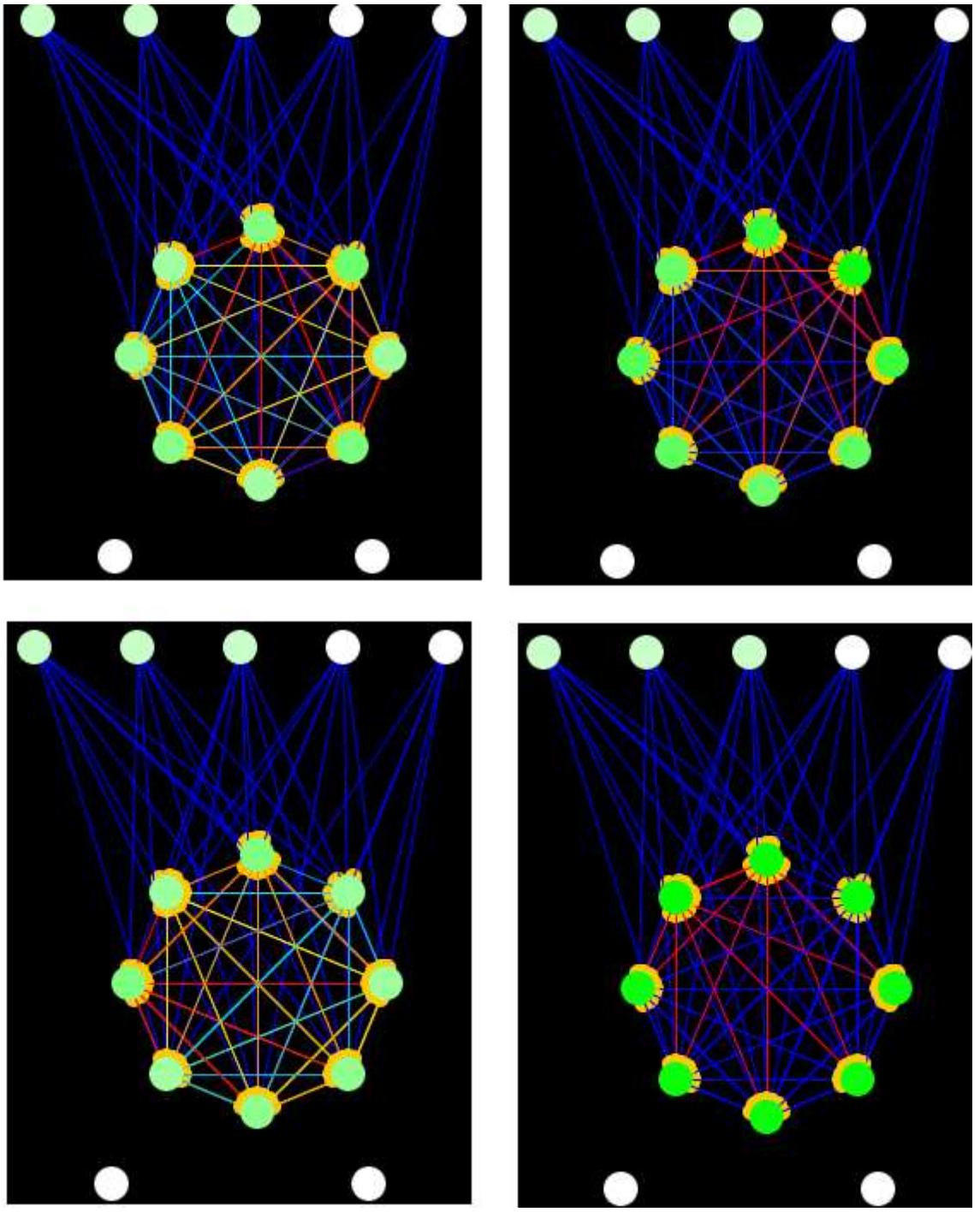


Figure 2.11: Testing the graph evolution to ensure stability.

2.3.2 Observations of the Real World

An exercise involving high school students presented an opportunity to investigate exactly how the simulation should behave, as well as giving some handcoded solutions to the line-tracking problem. In this exercise teams of students were required to code a model that would allow a Lego robot to follow a line on the ground. Several programs were created, but intrinsically they all shared the same design.

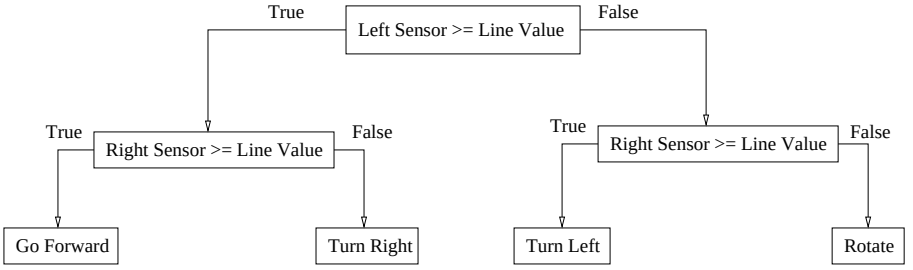


Figure 2.12: The manually coded design that most often came up.

This design enabled the network to follow a line by turning on the opposite motor to the sensor that saw the line, and was optimised such that if both sensors saw the line then both motors would turn on. The robot would also turn around when both sensors were off, thus allowing the robot to 'double back' along the line.

A similar design encoded in a neural network would use two input neurons cross-connected to two output neurons. This arrangement is known as a Braitenberg vehicle (1984 [2]). In this arrangement as the value of the left sensor increases, the speed to the right motor is increased and similarly for the right sensor and left motor. If both sensors see the same value then the motors will run at the same speed and the robot will go straight. Unlike the hand-coded version however a more complex network would be required to allow the robot to turn once it lost the line.

2.3.3 The computer world

The simulation consists of two main components. The first is the world which is broken into walls, background noise, and the goal line. The walls are simply restricted areas that the robot is restricted from entering, and are beyond the visibility of the robot. The background noise is a Gaussian distribution of light levels and covers the majority of the world except for the walls and the line. Finally the 'line' to be followed is some area that may be of a different light level to the background. This line is used for scoring the individuals.

The second component is the robot. The size of the robot is modelled based on its real-world counterpart and uses three physical inputs and two motor outputs. The sensors come in two variants. The first are based upon a ground facing light sensor with a visual radius of 4 cm. The second variant is a vision arc that extends forwards as would be expected if the light sensors were facing diagonally down. In both variants the light sensors can tell the light level of an area, but can't distinguish between the background and line (other than by their light levels). Several other types of sensory inputs were also provided and included a compass, wheel rotation counters, and scent-based sensors (similar to ant trails).

2.4 How the population was measured

The population is evolved using a tournament selection process. To begin the process some number of random individuals are created and allowed to run through the world so that

they may be scored. Once these individuals have been scored the process then begins to selectively breed individuals. The following steps are:

1. Randomly choosing whether to re-test an old individual (with a low probability), or breed a new individual (high probability).
2. If breeding a new individual then
   (a) Select a random number of individuals according to the tournament size, and choose the single fittest of these.
   (b) Repeat step 2a.
   (c) From these two individuals choose the individual with the highest fitness. This one becomes the main parent, and the other one becomes the replacement parent as described in section 2.2.3.
   (d) Take the new child, and run it in the environment to score its fitness.
3. Else select the best individual and re-test.
4. If population size is greater than some maximum value, then randomly select one of the lower fitness individuals and remove them.

Some points to note about the process are:

- The probability distribution for randomly selecting individuals is a Gaussian function over the individuals with the mean centered over the best individual. This means that the chance of randomly selecting an individual of high fitness is significantly greater than selecting one of lower fitness. This is used to both select the individuals required for breeding, while its inverse is used to remove poor performance individuals.
- Re-testing is performed at random over the highest performing robot to help eliminate robots 'getting lucky.' There are often cases where a robot will be placed in a position that will naturally give it a high score. This may result in the robot being rated above its normal abilities. By re-testing the best individual the chances of it remaining in the top position due to luck are diminished.
- When the population reaches are certain maximum, a lower performance individual is selected by chance to be removed. This is done rather than removing the worst case individual to prevent the population from converging too quickly. By allowing slightly divergent individuals to remain they help keep certain attributes in the gene pool alive. These can sometimes be used at a later stage to bypass a stalemate in the evolutionary process.

2.4.1 The fitness function

The fitness function is calculated as:

$$\begin{aligned} areaLine &= number\,of\,pixels\,on\,line \\ areaRobot &= number\,of\,pixels\,on\,robot \\ fitness &= \frac{areaLine}{areaRobot} \\ normalised &= \frac{fitness}{1 - fitness} \end{aligned}$$

This function is designed so that the individual is only rewarded once for travelling over a given section of the line. This prevents individuals from evolving to simply travel in circles over the line and to prevent individuals from scoring points for repeating small sections of the line. If desired, it was possible to designate a desired end point. In this case the fitness score was divided by the distance to the end point.

2.5 Putting It All Together

Several maps were built that each contained different basic qualities, such as curving left lines, curving right lines, sharp curves, and right angle and intersecting lines. Then for each of these maps the program was left to evolve and test the individuals. Finally, once a suitably fit individual was found, it was saved and analysed.

Chapter 3

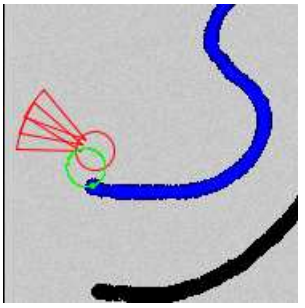
Results

In this chapter a presentation of how the evolution handled searching for the individuals, as well as a showcase of some of the individuals and their behaviours is given. An investigation into the relationship between the environment and the types of individuals that were found in each is also given.

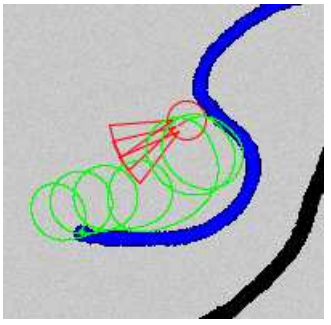
3.1 The Stages of Evolution

The evolutionary process produced similar sets of individuals at certain generational steps. These results show that the world successfully rewarded partial solutions and thus gave the population a smooth fitness surface to climb. These behaviours could be classified into four stages:

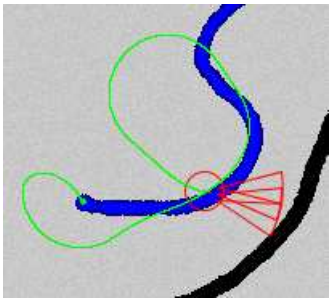
- **Stage One**: These are the youngest individuals and generally either stay in one spot, or rotate around one motor. Of these the rotating individuals gain the highest fitness.



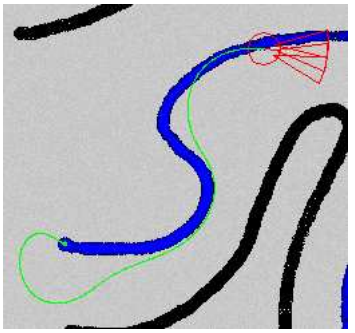
- **Stage Two**: The later generations individuals consist of straight runners who travel in a straight line regardless of the sensory inputs, evolved spirallers (shown below) who will rotate if not on the line, but travel forward otherwise.



- **Stage Three**: By stage 3 individuals are capable of turning in both directions to follow the line, but will favour a tendency to turn in one direction. This results in individuals that will repeatedly follow some section of the line, but not the entire line.



- **Stage Four**: The final generations are capable of fully following the line. These can be divided into a group that turn around at the end of the line, and those that continue travelling straight after losing the line.



3.2 A Zoo of Critters

Several solutions were found that were able to perform some aspect of scoring points. The types of individuals found were generally dependent on the type of map being used as each map favoured different types of behaviour. Generally, as with the evolution stages in section 3.1, these maps could be broken down into categories that showed the type of individual that was generated. These, along with the generic behaviour of the individuals, are discussed in this section.

3.2.1 Basic Curve

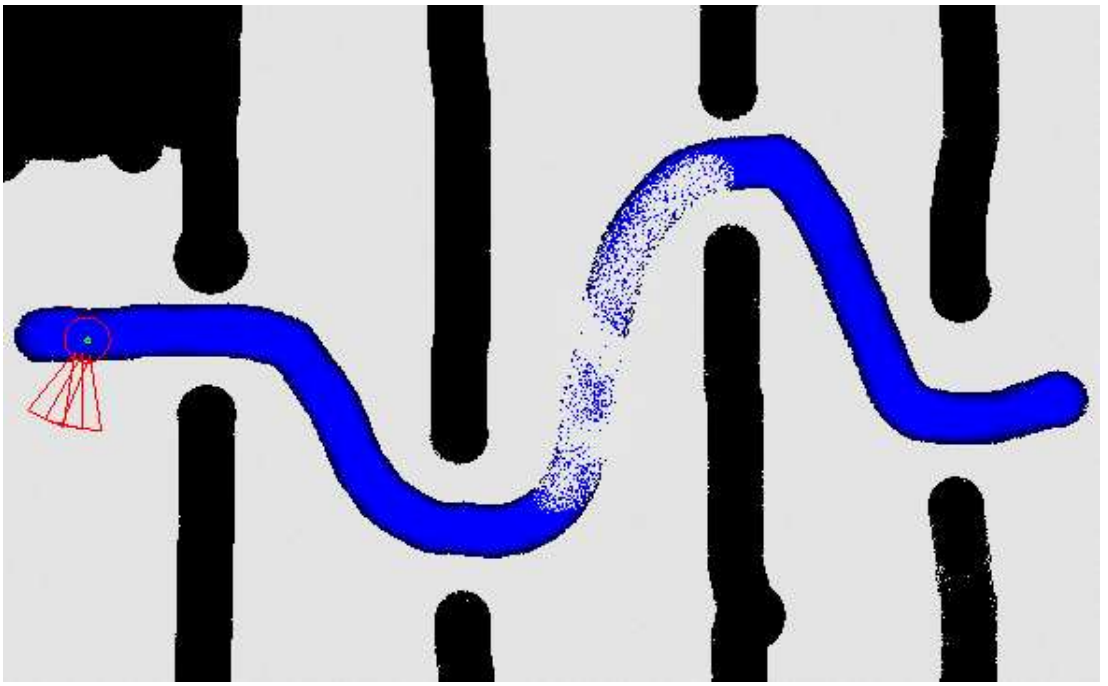


Figure 3.1: The basic map with a track larger than the robot bodies

Figure 3.1 is the basic map used to initially test the abilities of the individuals. It features a large track that very gently curves and is wider than the width of the individuals body. The track becomes broken after a short distance. This was designed to give some individuals the opportunity to develop other features that may allow them to traverse across such broken areas.

Those individuals that were found in this map were simplistic and easily found by the evolutionary process. Figure 3.2 shows an example of an individual (and its trail in green) that navigates the path (in blue). Characteristically the individuals on this map tend to run inside the line and bounce off the edges of the line. Likewise, when looking at the evolved network, and then the sliced network it is easy to see that the individuals do not require complex networks to solve the problem. For example, the individual in figure 3.2 (whose network is shown in figure 3.3) only uses one sensor to navigate.

Sadly none of these individuals were ever capable of finding the second half of the line. The individuals who did find this line only ever did it by chance, and could not repeat the event. So because these individuals were unable to be duplicated, any characteristics within the network that may have aided finding the second half of the line were never carried in the gene pool.

3.2.2 Tight Curve

Figure 3.4 is a more complex map that uses lines smaller than the robot bodies width, and a tighter curve. This curve is sharper than the sensors resolution (the robots would reach the line perpendicularly and hence the sensors would all see the line simultaneously).

This map caused more of a problem as the individuals often became stuck on the second curve. A generic characteristic of these individuals was the double backing described as stage 3 evolution. This map also spawned the more complex individuals with very few individuals sharing a similar graph structure.

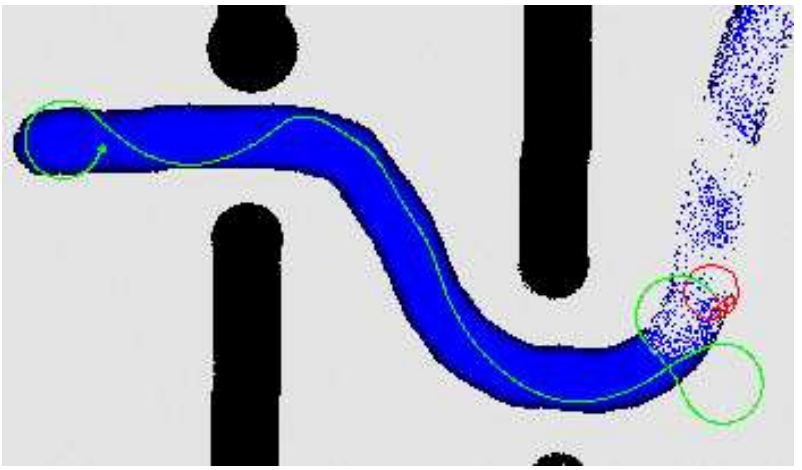


Figure 3.2: Individual navigating the map

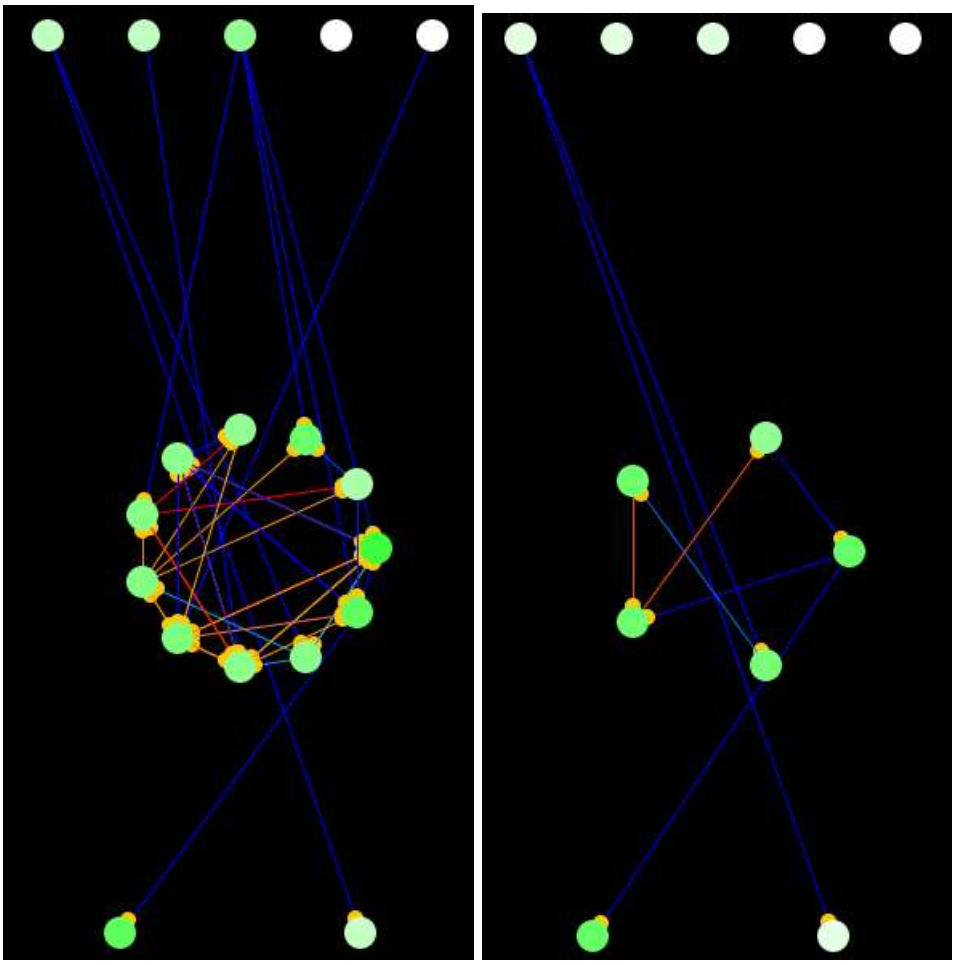


Figure 3.3: The network of the individual shown before, and after slicing

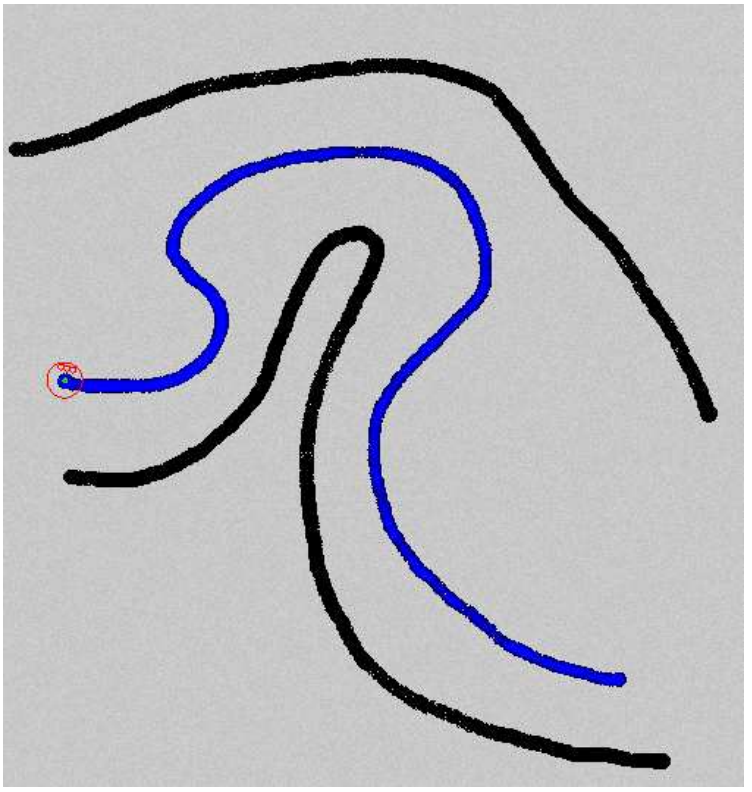


Figure 3.4: A map with a smaller track and tighter curves.

As an example of the difficulties found in this map, the individual demonstrated in section 3.2.1 was run in this map several times. The results and its trails are shown in figure 3.5.

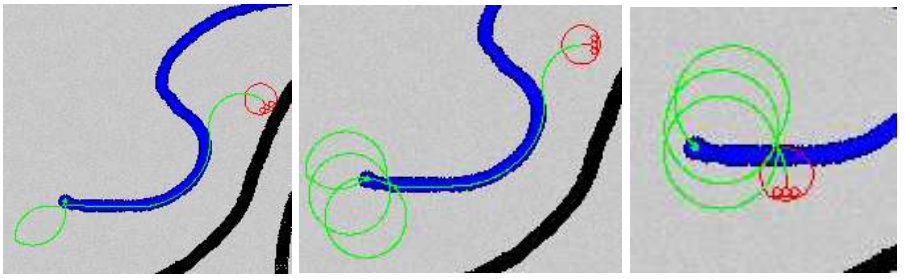


Figure 3.5: The individual from the basic map attempts to solve the tight map.

Some of the more effective individuals used a similar but more advanced search strategy to the individual above. These individual's trails are shown in figure3.6. One of these individuals is also interesting because its able to follow various types of lines. This individual is shown navigating both the basic map, and another map that contains right angles in figure 3.7

3.3 Whats Under the Hood

3.3.1 Slicing the Brains

The main analyses of the networks was to see if there was any inherent underlying structure that would be similar across various models. For instance, did each of the individuals end up using a modified Braitenburg vehicle?

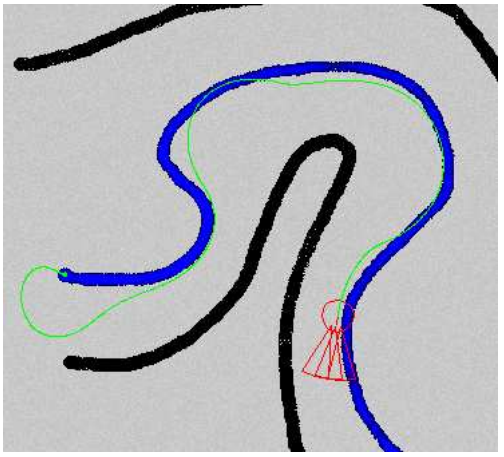


Figure 3.6: The more advanced individuals and their trails.

To do this analysis the networks were 'damaged' to see if one could reduce the size of the networks while still maintaining their evolved fitness. The process for doing this is fairly simple:

1. Begin by storing the individuals current tested fitness.
2. Repeat:
   (a) Select a random number of links to delete from the network.
   (b) If a node has no external links then remove the node from the network.
   (c) Test the 'damaged' individual.
   (d) If the fitness score is approximately equal than use this individuals network.
   (e) Else try again.

This method of slicing the network was chosen over some other more methodical searches because of a few possible network characteristics that would be hard to identify. For instance, looking at network traffic amongst the nodes was not viable. If one considered that two nodes may be sharing a task, but could themselves be capable of doing the task without the other node, then each node would have similar traffic levels but would still be redundant.

Step 2a was also necessary to handle 'messy' residual subgraphs such as the sub-structure within the network that performs a redundant task. By removing a link within this sub-structure the area becomes unstable and starts interfering with other areas of the network and hence lowering the fitness. By being able to remove several links in one go the chance of completely removing a redundant subgraph is increased.

3.3.2 Watching the Network

After slicing the network to reduce the complexity, the next step was to watch exactly what the controller was doing. To do so a record of the network's the output of all its cells was logged as it performed runs across the maps.

Once the log was recorded the patterns were graphed and examined to see how the network was arranged. Some characteristic features were especially sought after in these graphs. If several nodes were mirroring the activity of other nodes then we could conclude that they were necessary (i.e. weren't removed by the slicing) they must be amplifying

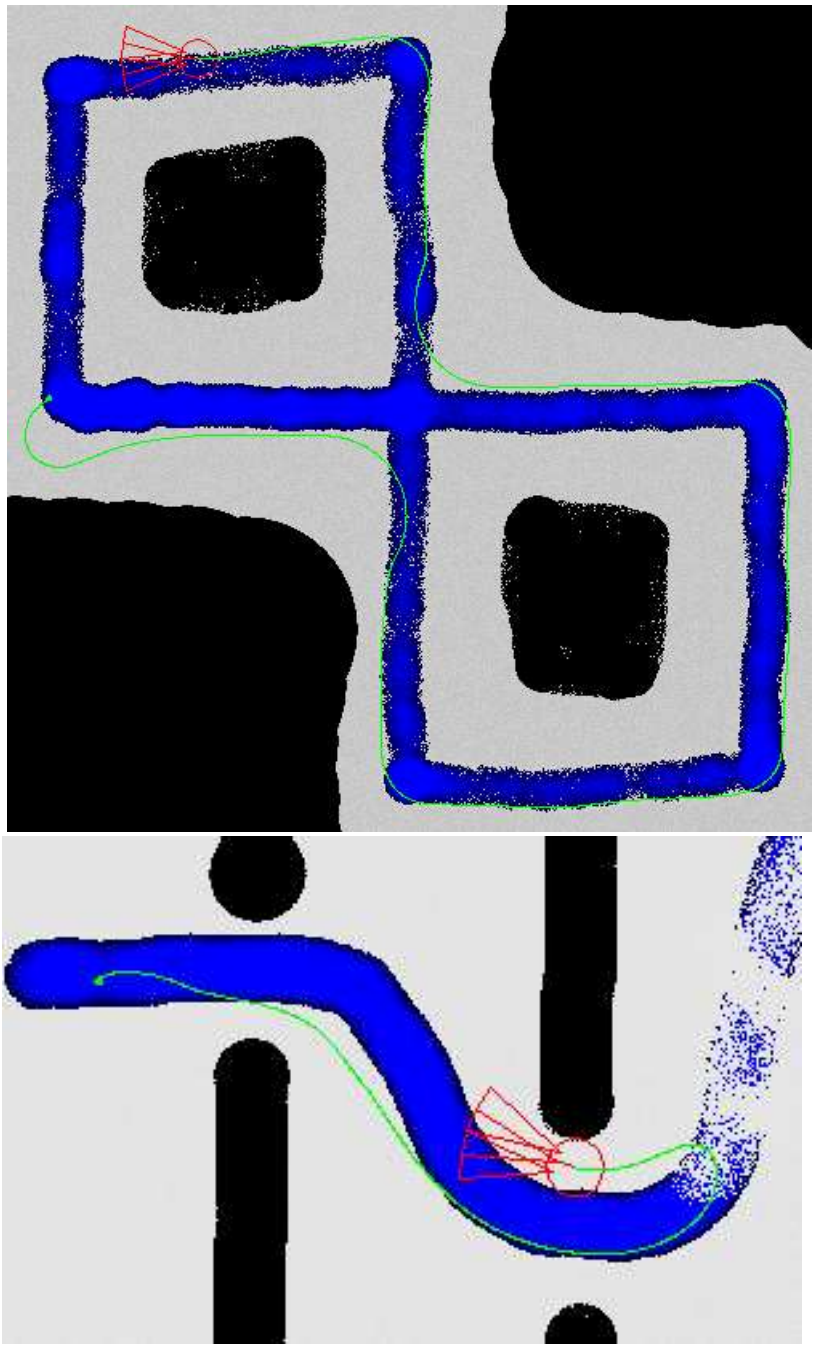


Figure 3.7: The advanced individual travelling being tested on other maps.

the signal. This could be further shown by large humps being connected to several mirror nodes.

A more interesting arrangement would be some cyclic pattern in the graph, especially if shared by several nodes. This would indicate that the network is capable of utilising a time component when navigating.

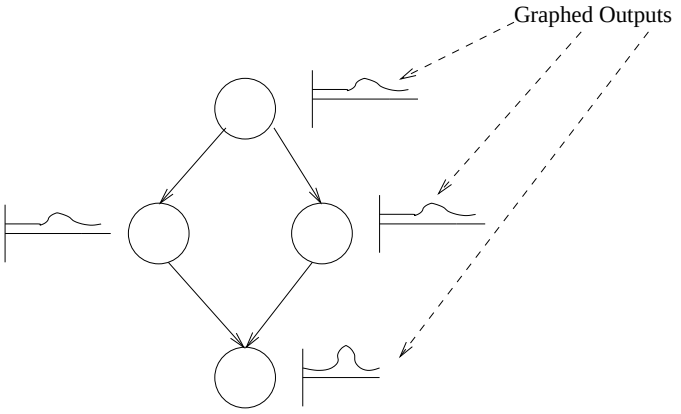


Figure 3.8: A network layout with two mirror nodes amplifying the input

After observing the behaviour of the individuals, the next thing of interest is to take a look at the internal behaviour of the neural networks. In this phase the network was sliced to the minimum number of required nodes to maintain the individuals fitness, then the network's activity was logged as the program ran. This was done by recording the output level of all the nodes at each time cycle.

We begin by looking at the individual used for the Basic map. As pointed out this individual only used one sensory input to navigate the line. When viewing the graph of its networks activity we see that the individual used a 'memory' layout to produce a constant output for one of its motors, and directly mimic the sensor for its other output. The trail has already been shown in figure 3.2 so only the graph is displayed in figure 3.9.

Careful observation shows that the network produces a three node memory circuit that is set when the individual is born, but remains constant after that. This leads the individual to continually move in a circle with one motor at a constant output and the other motor turning on and off depending on whether it is on the line or not. In the zoomed in section we can see the pattern of node activity within the first 50 iterations as the network self adjusts itself. These patterns are setup from the networks structure and learning rule.

A more interesting individual was spawned from the Tight Curve map. This individual uses a cyclic timer that fires the right motor at regular intervals. The amplitude of this motor is still affected by the internal processing nodes, but the left motor uses a combination of two inputs to deduce its output. Again like the basic individual, this one has an initial adjustment period of approximately 50 iterations.

One of the advanced individuals from the Tight Curve map has its brain log shown in figure 3.11. The figure shows a close correlation between the inputs and outputs. This suggests that the individual uses a modified Braitenberg design. One motor is directly attached to the opposite sensor input, while the other motor has some logical processing between the input and the output. Consequently this individual is capable of turning about once it finishes following the line and can then follow the line back.

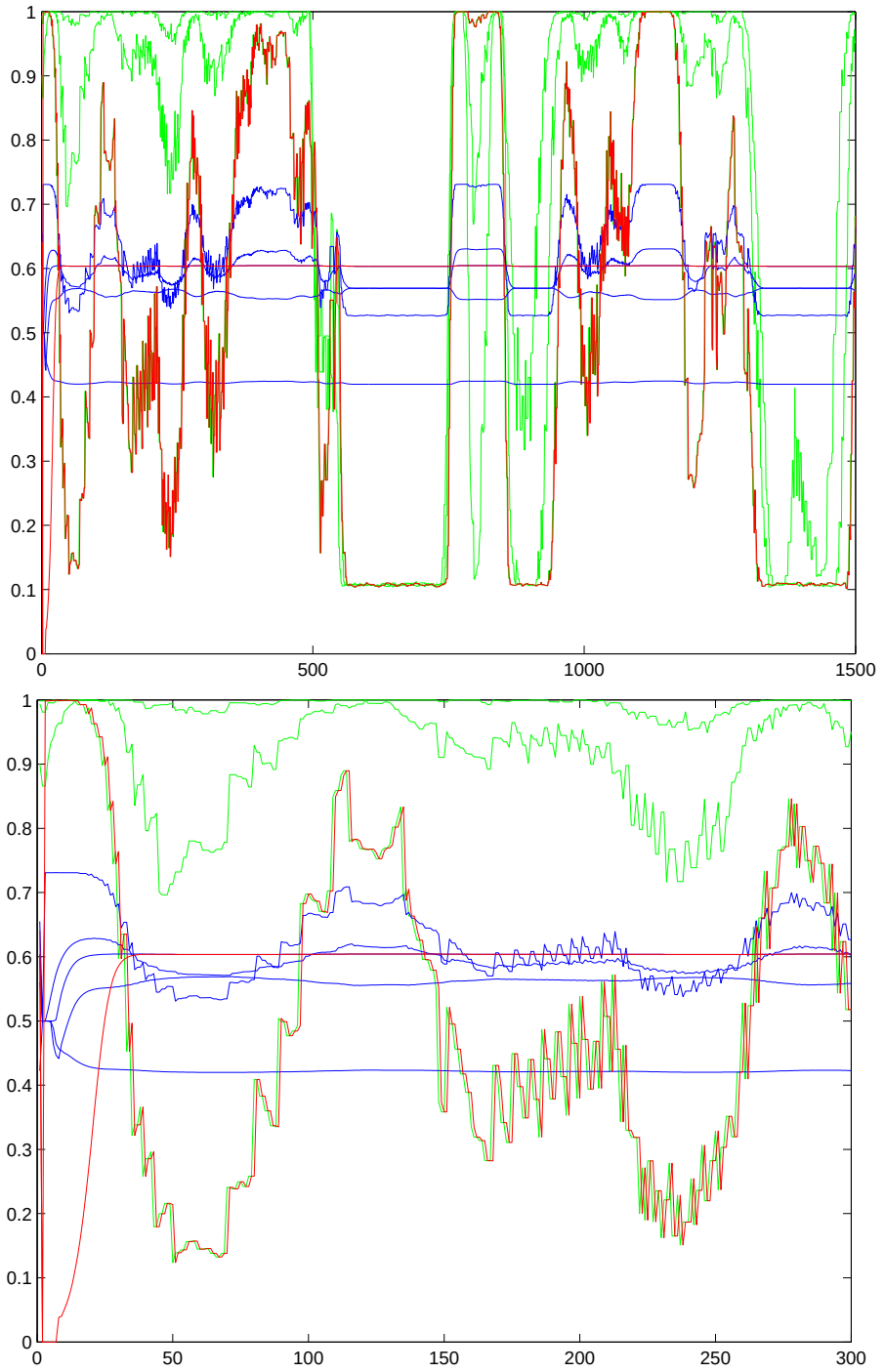


Figure 3.9: The brain activity for the individual. The second graph shows a zoomed in section of the first 300 iterations.

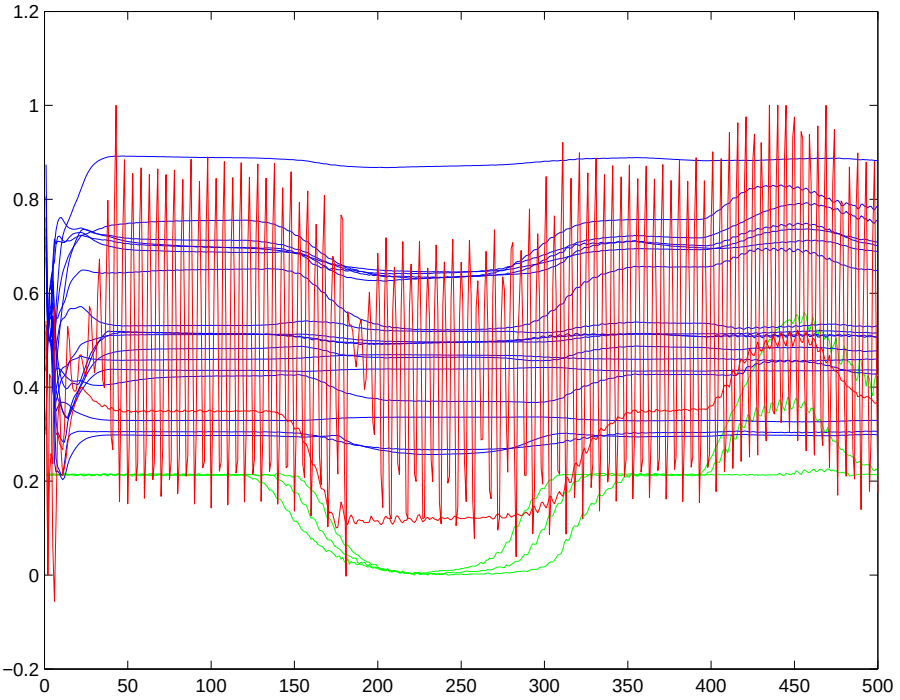


Figure 3.10: The brain activity for an individual using a cyclic timer.

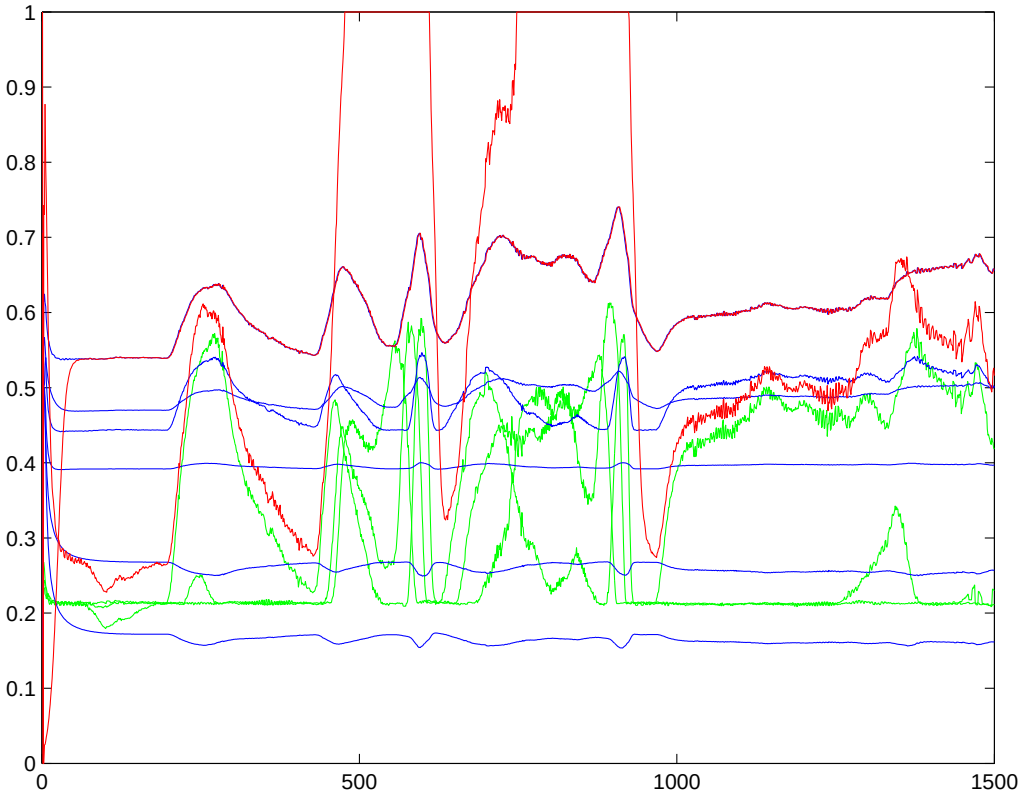


Figure 3.11: The brain log activity for the advanced individuals.

Chapter 4

Discussion

4.1 So What Exactly Happened

One of the motives of this project was to remove the specification of the network's structure from the researcher, and instead leave it up to the algorithm to determine. The intention was that by doing so the evolutionary process would find some structure that would increase the network's adaptability to other environments. However the final individuals were not as adaptable to other environments as might have been hoped. Rather like the reactive parameterised controllers of the standard approach, individuals suffered from over-fitting to their native environments. They became optimised to handle the environment that they were raised in, but did not perform as well in any new environments. A similar effect took place when the sensors were changed from directional downward light sensors to forward facing light sensors (or back). The individuals did not adapt to the different characteristics of these sensors.

However, using these individuals as the seed for a new generation of individuals in the new environment proved to work relatively well. As long as key components were available within the individual these could be utilised in the individual's children. For instance, if an individual was able to follow thick lines, then in the thin lined map it would fail to turn the sharp hair pin properly. Instead it would often end up double backing along the line. A similar thing happend when an individual was raised on a thin lined map and moved to a thicker lined world. Here the individual would follow the outside of the line.

4.1.1 Did They Employ Lifespan Learning?

The large problem with these individuals is that they did not use any input based learning techniques to solve the problem. This means the controllers were designed to set their weights to specific values rather than designing an learning rule that took into account the state of the world. It became easy for individuals to encode a mechanism that would set up the parameters of the network to solve the problem, however these parameters would arrive at the same values when the individuals were moved to a new environment. So though the entire population could converge on a solution, the individuals themselves were not adjusting to the environment.

The main factor that contributes to the individuals opting for an encoded behaviour rather than a learnt behaviour is known as the Baldwin effect. This is described as:

> "In an evolving population, individuals that are able to acquire a trait during their lifetime will be selected if it is advantageous. The offspring of these learning individuals will then come to dominate the population since it is their genes

> [that] are passed on to later generations. Once the population is full of learning individuals, the learnt trait may [be] genetically specified in subsequent generations."

That is, in the population any individual that is successfully capable of learning some characteristic of its environment will be selected for breeding. The trait for that type of learning will be located somewhere within its genetic structure. Eventually the trait will become hard-coded into the genetic code of the individual, and that individual will no longer 'learn' that behaviour. The result is that the individual will become optimised to suit whatever environment it is raised in.

4.2 What Were They Really Capable Of

DNA ran many iterations and from these a set of individuals were found that were able to successfully track the lines. However some of these individuals showed signs of more interesting behaviours. In general the successful individuals solved the problem by performing some logical mapping from the inputs to the outputs whereby the individuals relied purely on their current input to deduce what their output should be. However a select few of these individuals were capable of demonstrating behaviour that utilised a timing component to solve the problem.

There were also a handful of individuals who were able to take this timing and use it to perform a set of actions. For example one such individual moved for a set distance and then made a slight turn. This individual was evolved from the Food map and so this may have been a useful adaptation to perform large circular searches.

These traits point to the networks being able to perform more complex operations other than simple logical mappings from the inputs to the outputs. By moving the individuals into a more complex domain and producing a more efficient and effective means of searching for these individuals, these individuals could probably show far more interesting behaviour.

4.3 How Were They Limited

The limitations of these individuals arose from the limitations of the environment they were contained within and the method used to configure the networks. This can be explained simply by looking at how closely the interactions of animals and their environment in our world are, and the exact mechanism by which evolutionary processes work.

4.3.1 The Effects of the Environment

The complexity of an environment plays a strong role in the complexity of the individual that is required to survive in that world. This theory is known as Umwelt[5] and describes the evolutionary advantages to leaning heavily on the environment to provide the individual with models and predictions. It also suggests that creatures within the environment will 'choose' to use only a limited set of possible sensory inputs to describe their world.

For instance, a flea need not be able to see an approaching dog. It only needs to know that somewhere nearby is a warm bodied (and possibly smelly) object that is within jumping distance. Likewise, for a simpler creature such as a plant, all the sensory input it needs to know is which way is up, and from which direction the sun is coming. Other possible inputs the world may provide such as sound, colour vision, etc can simply be ignored.

This is apparent in the line tracking world. Conceivably there are only four different states in which one can exist in. You can be on the line, partially on the line (to the left or

right), and off the line. In the three states where you can see the line the solution is trivial, however when you're not on the line what exactly can you do? With no mapping of the world any solution is almost equally viable.

In some individuals compass and wheel rotation counters were added as sensors. When the individuals were forced to start off the line they would eventually discover how to navigate to find the line. However once these individuals loose the line they reverted to their previous search strategies. They simply did not remember that they had once seen the line.

Some further evidence that shows the individuals dependence on their environment was found when attempting to move the individuals between differing environments. Though they were marginally successful, for the most part they failed to compete with even the simplest solutions. The individuals had evolved to take advantage of quirks in their native environment and when these were removed the individuals could no longer perform as well. However, individuals who remained within their environment proved very robust. Dissecting the individuals showed that they used an amazing array of redundant and backup subgraph structures that could each perform some part of navigating the line.

4.3.2 Evolutions Search

The evolutionary approach to organising networks had some flaws that lead to imperfect solutions. These problems can be summed up as a lack of divergent evolution, non-smooth fitness gradients, and an inability to recognise useful sub-components from the neural nets.

Divergent evolution is the essence of the abundance of species in the natural world. It allows a population to divide and exploit differing resources within the environment (niche finding). These divided populations then continue along slightly differing evolutionary tracks, thus allowing nature a way to investigate multiple solutions.

The populations in DNA lacked any sense of 'species' that would prohibit vastly differing solutions from mixing. So in all cases the populations would converge on the most successful individual within that population. In scenarios where the best individual in the population was not the best solution, the population would become stuck at a local optimum without having the genetic material to bypass it. Ideally within a population each possible sub-solution should be investigated by creating a new breed that follows some specific evolutionary path. This would also allow less efficient genetic material to remain in the populations without becoming completely lost as with the convergent evolutions.

Genetic algorithms need a smooth fitness gradient that can be climbed through sub-solutions. In section 3.1 this slope of sub-solutions was shown. The problem is when a more optimal solution does not lie perfectly on the slope then the algorithm can only find the solution by chance mutation. Even with the mutation the first individual may be too fragile to repeat the task on a regular basis, and therefore fails to become accepted as the new best.

As an example, consider the Basic Curve map. In that map individuals are trained to follow a large line, and so end up producing a curving behaviour that makes them 'bounce' off the edges of the line. However once they reach the end of the line this curving behaviour makes them turn in a circle until they find the same section of line again and so prevents them from finding the other half of the line. Any individual who finds the other half gets a lucky starting position. When the individual's children are then tested they either get a different starting position, or have some change to their network that prevents them from replicating their parents behaviour. So, though the individual may be fitter, its fragility prevents it from becoming a viable solution, and it ends up being discarded by the population.

Alternatively the population may not have enough experience to create a sustained set of solutions. As an extreme example, if the starting point is randomly determined on a large map with scarce locations of line, how would you learn that being on the line is what

scores you points. This effect is apparent on the Food map. Individuals evolved on this map usually ignore any sensor inputs regarding food and focus more on generating a large circular pattern with corrections from the outside wall. Since the individuals only rarely encounter the line they do not have enough information available to learn that the line is whats providing them with a score. Rather they continue to believe that travelling in a large circle is what scores them points.

4.4 Could Anything Be Done to Make Them Better

The study of these individuals leads to the question of whether or not the DNA concept is fully tested by the environment described here, or is it capable of doing more. The arguments presented above show problems with both the environment used to test the individuals, as well as the mechanism used to search for them. It makes sense that these are the two areas that improvements can be found.

4.4.1 Environmental Improvements

As already stated, a complex environment leads to a need for more complex individuals within that environment. Therefore if we wanted to investigate more interesting behaviour from this method of building networks, then we would need to build a world that is more complex, and more suitable to nurturing such artificial life. As an extension to this project, the world could be extended to simulate motor vehicle driving using the neural networks evolved here as the steering controllers. This could either be augmented by a decision planner that would provide directions (turn left, go straight etc) to the network, and the network would handle short range object avoidance and staying on the track, or using multiple expert networks that compete to be allowed to execute their outputs.

4.4.2 Evolution Improvements

Improving the evolution engine might come as two parts. Firstly refinements to the graph evolution approach that would allow all the mutation effects to be reproduced using only cross over (such as forming new connections from inputs to outputs) would improve the engines simplicity. Secondly, a means of causing divergent evolution through either spatial locality (i.e. proximity of fitness scores) or similarity of graph structure would allow the evolution engine to search multiple possible solution paths at once.

A concept of using co-evolution whereby two creatures evolve in competition to each other, could be applied to improve the individuals within the world by exposing them to more difficult tasks. This comes as an interesting idea of applying evolution to the environment such that it evolves against the fitness of the individuals. For example, an environment where the average fitness of the population is high, would score lower than an environment where the average population was lower. Presumably then individuals will have to continually evolve to become more suited to changes within the environment.

4.4.3 Controller Improvements

Improvements can also be made to the controller design. As mentioned earlier, one of these improvements would be to use multiple expert networks, or to cluster a current network into functional subsystems. This would be something similar to a human brains functionality were partitions in the brain are responsible for different tasks. From the networks

perspective partitions in the network would focus on one role and would be able to influence other partitions based on its own 'opinion.' To provide an example, one partition may be responsible for being able to turn left, turn right, and go forward, whilst another partition takes sensory input and decides which direction to turn (which is then passed to the lower navigational partition).

Also, in continuing with the theme of unrestricted neural architecture, nodes within a network may be able to continuingly adjust their connections to other nodes. This process may be similar to a biological brain in that nodes within the network would be able to create new connections (and new nodes) during the individuals lifespan. The rate at which they form these connections would be dependent on the activity of the respective nodes in such a manner as to be useful in storing knowledge about the state of the world.

Chapter 5

Conclusions

A growing amount of research interest is focused on self arranging dynamic networks. These systems seek to build networks that stabilise on a solution through the use of customised learning rules. This is in opposition to standard systems that use a generic learning rule to update the parameters of the network. However, research shows that the dynamic systems used in this report are more biologically plausible than the traditional approaches.

The project set out to build a Skinnerian creature; a creature capable of learning to behave in its world. We use the domain of line following as a means of testing the networks that were created. This domain is a common behaviour trait in nature with many species displaying aptitudes to various levels of line tracking. These range from ants following layed out scent trails, to birds navigating from landmarks. The domain also provides a simple simulation for practically applying this research to driving vehicles on factory floors, or even driving on roads.

Floreano showed that a dynamically updating neural network could function. This neural network evolved its learning rules by specifying the learning components of each of the neurons within the network. However Floreano used a statically structured network of 12 nodes in all his work. This report showed that Floreano's work could be augmented so that both the learning rules and the structure (both size and connectiveness) were evolvable. This allowed the evolution engine to search for neural networks that could solve a problem without any fixed restrictions on the networks structure.

While doing so, a method of applying genetic algorithms to cyclic graphs was found. This report showed that this system was both simple, effective, and had the ability to subtilely change the networks. This differed from other approaches where minor changes to the genome of the graphs caused major changes to the graph structures.

The individuals that evolved from using the hybridised approach were studied to determine how viable this method is for situated robotics. We discovered that the approach is very good at solving the task of line following, but has a problem adapting to changes within the environment. This was followed by a discussion as to why these individuals are limited and what can be done to improve the capabilities of the individuals.

Bibliography

[1] AL GLOBUS, J. L., AND WIPKE, T. *Automatic Molecular Design Using Evolutionary Techniques*, vol. 10. September 1999.

[2] BRAITENBERG, V. *Vehicles: Experiments in Synthetic Psychology*. The MIT Press, Cambridge, Massachusetts London, England, 1984.

[3] CHALMERS, D. *The evolution of learning: An experiment in genetic connectionism*. Proceedings of the 1990 Connectionist Models Summer School,, San Mateo, California, 1990, pp. 81–90.

[4] CONRAD, M. The geometry of evolution. *Biosystems*, 24 (1990), 61–81.

[5] GUDWIN, R. R. Umwelts and artificial devices - a reflection on the text of claus emeche: Does a robot have an umwelt?

[6] HEBB, D. The organisation of behaviour, 1949.

[7] JAKOBI, N. Half-baked, ad-hoc, and noisy: Minimal simulations for evolutionary robotics, 1993.

[8] LAWTON, J., AND WIPKE, T. Graph crossover. Tech. rep., Al Globus, CSC at NASA Ames Research Center Sean Atsatt, Sierra Imaging, Inc., 2000.

[9] NOLFI, S., ELMAN, J. L., AND PARISI, D. Learning and evolution in neural networks. Tech. Rep. 9019, 1990.

[10] URZELAI, J., AND FLOREANO, D. Evolution of adaptive synapses: Robots with fast adaptive behavior in n ew enviroments, 2002.

[11] WILLSHAW, D., AND DAYAN, P. Optimal plasticity from matrix memories: What goes up must come down.