Project Report

Name: Christopher Davenport

Degree: Computer Science & Artificial Intelligence

Department: Informatics

Candidate No: 91067

Title: A Simulation for Comparing Training and Evolving of Agents

Supervisor: Chris Thornton

Year: 2006 - 2007

Statement of Originality

This report is submitted as part requirement for the degree of Computer Science and Artificial Intelligence at the University of Sussex. It is the product of my own labour except where indicated in the text. The report may be freely copied and distributed provided the source is acknowledged.

Signed: _____ Dated: _____

Acknowledgements

Many thanks to Chris Thornton for the help and advice given over the last year and to all my lecturers who taught me the concepts needed to perform this project.

Abstract

The project compares the training of neural networks, through genetic algorithms and backpropagation, to be used as neural controllers for artificial agents in a simplified game of capture the flag. The game involves two teams of agents, several obstacles and two coloured flags. Each team must attempt to reach the oppositions flag, pick it up and bring it back to their own base. The team that does this the largest number of times wins. Each agent consists of a number of sensors that act as inputs to their controllers which produces a set of outputs to control the left and right wheel speeds allowing the agents to move. One team was trained using the genetic algorithm approach, implementing a roulette wheel selection, and a form of subsumption architecture to split the tasks of collision avoidance and flag finding into separate, smaller networks. The other team had a single neural network which was trained using backpropagation from input-output data gathered from a hand-coded agent. It was discovered that the subsumption architecture being used for the first team needed some alterations in order for it to work properly and the causes of these problems were discussed. The teams were tested against each other in 3 different environments/ maps and overall the GA team performed significantly better, with overall much better collision avoidance behaviour and less occurrences of getting stuck against obstacles. The results also showed that the genetic algorithm was able to produce an overall simpler controller with significantly less network connections than the one produced using the backpropagation technique. The discussion determined that part of the problem with the backpropagation approach was the need for pre-existing data in order to train the network and that the data came from an agent that wasn't particularly good at collision avoidance anyway, highlighting the difficulty in trying to manually program an agent based simply on sensory input values. Other issues are also discussed but the overall conclusion was that the GA technique performed a lot better, but also highlighted the importance of correctly setting up the fitness functions and environments in which the genetic algorithm performs. Due to the potentially vast nature of possibilities of this project many extensions and improvements have also been proposed.

Contents

1. Introduction

1.1 Project Overview

This project involved the development of a computer simulation of a Capture the Flag game in java, which allowed techniques such as Backpropagation and Genetic Algorithms (GAs) to be compared when it comes to training the agents for the task. A game of capture the flag consists of two teams and two flags (one for each team) in an environment which consists of various obstacles. The aim was for each team to try and "capture" the opposing team's flag by finding it and bringing it back to their own base.

Each team was made up of a number of virtual robots (agents) each with a series of input sensors, two wheels and a neural controller which converts the sensory inputs into motor outputs. The neural controller consisted of a series of layered artificial neural networks (ANNs) implemented in a similar way to Brooks subsumption architecture [2], where each layer controls a different behaviour. The simulation program allowed a neural controller to be trained for each team using either a genetic algorithm or backpropagation and a tournament environment to allow the teams to compete to see which method performed the best.

The first goal of the project was therefore to build a fully working simulation environment in java which could be used to train neural controllers using genetic algorithms and backpropagation with a variety of parameters, and also allowed these controllers to be tested in the environment. The architecture of the sensors and neurons in the neural controllers was another important factor that had to be carefully decided upon in order to get the best possible performance for each of the techniques. The actual comparison between the two techniques was achieved by training and evolving the neural controllers for a specified number of epochs/ generations and then placing them in a tournament environment, where they competed against each other. The performances of the two techniques was then analysed.

To make the process easier each stage was broken up into several smaller stages. The neural controller being evolved by a genetic algorithm was not evolved to do everything at once. Instead, each behaviour was evolved one at a time, gradually building up to the desired overall behaviour, similar to [4, 11, 12]. Similarly the process involved training the agents for games of gradually increasing difficulty. First of all the agents were trained to simply find the opponents flag without crashing into anything. The next stage was to not only find the flag, but to also return it to their own base. The final version, which was not attempted due to lack of time, was to build on the second one by attempting to train the agents to chase any opposition agent that was carrying a flag.

This report is broken up into several sections. The first sections talk about the motivations behind the project and provide an overview of the various techniques that were mentioned above that have been used throughout the project. The main body of the report covers the design of the system, a more in-depth description of the environment and the agents, and the experiments, along with their results and analysis. The final part of the report covers what has been achieved throughout this project and how the project could be extended. The project adhered to the professional considerations set out before starting the project (see Appendix A).

1.2 Project Motivation

Different algorithms and techniques are best suited to different tasks. A technique should be chosen based on its suitability for solving the problem at hand and often it isn't obvious which technique is most appropriate. Part of the motivation behind this project was therefore to determine, within the field of game playing simulated robots, whether it was easier to use a genetic algorithm to develop a suitable solution, or attempt to teach a controller how to do it by using pre-existing data.

Using a game of capture the flag may appear uninteresting and rather simple; however this is not necessarily true, and it has proved a popular basis for a lot of research [7, 8, 13]. Even relatively straightforward tasks, such as collision avoidance are rarely as straightforward as they seem. For example it may first appear that using a simple cowardice[1] Braitenberg vehicle setup would create the desired collision avoiding behaviour. However, in the case where a wall is directly in front of the agent, it would accelerate directly towards the wall rather than turning away from it. Developing a neural controller to do this is therefore quite complicated, and it is not until we have a good grounding of these more simple tasks, can we start to develop more complicated control systems.

Producing fitness functions for a genetic algorithm is no simple task, and is made even more difficult if the fitness function should incorporate several different tasks, as in a capture the flag game. If a poor fitness function is used then the solution produced by the genetic algorithm may only perform part of the desired behaviour, or not perform at all. This project, as mentioned, attempts to overcome this by using a technique similar to subsumption architecture [2] to break down the genetic algorithm approach into separate layers allowing simpler fitness functions to be produced for each layer. Another motivation for this project was therefore to see how well this approach worked, and to see if it made the process of evolving behaviours any more straightforward.

Genetic algorithms and supervised learning techniques like backpropagation have several parameters that can drastically alter their performance and outcome. Parameters are very much problem dependent in that a specific set of parameters will increase the speed for one task but not necessarily another. Finding the right parameters for a task can be very time consuming and some techniques will react vastly differently to slight changes in the parameters. Which technique, GAs or backpropagation is less susceptible to slight changes in parameters, and therefore more likely to find a good result?

The main method for comparing different problem solving techniques is to see how they perform individually and then compare them. By turning this into a simple game allowed to not only test the two techniques individually but also put their solutions in an environment at the same time, forcing the techniques to compete with each other directly, allowing additional comparisons to be made.

[1] A cowardice Braitenberg vehicle setup has 2 sensors and 2 motors where the left sensor has an excitatory connection to the left motor and the same applies to the right sensor and motor. This means that if it senses something to the left it will speed up the left motor causing the vehicle to turn to the right.

2. Background

2.1 Artificial Neural Networks

Artificial Neural Networks (ANNs) borrow ideas from their biological counterparts. Neurons in the brain are typically composed of a cell body (soma), dendrites and an axon. The dendrites provide the input to the neuron in the form of an electrical signal. If this input signal goes above a certain threshold the neuron fires, sending an electrical signal down its axon. Neurons communicate with each other across synapses. This is where the axon from one neuron transfers its signal to dendrites of other neurons. Each neuron can be connected to thousands of other neurons in this manner, producing a network.

An artificial neuron is very similar in principle to a biological neuron. It has a number of weighted inputs which are summed together producing the neurons activation. A function is then applied to this activation, with the simplest being a threshold function, producing an output. In essence an artificial neuron is an input-output computation box (black box), where given some input it produces a particular output.

An artificial neural network is a group of connected neurons, most commonly organised into layers. The most common network is one with 3 layers; an input layer, a hidden layer, and an output layer (see Figure 1). Many papers do not consider the input layer as a layer in the network, so refer to this as only a 2 layer network. For clarification purposes the rest of this paper will include the input layer when mentioning a neural network. The input neurons are simply the input values to the network.

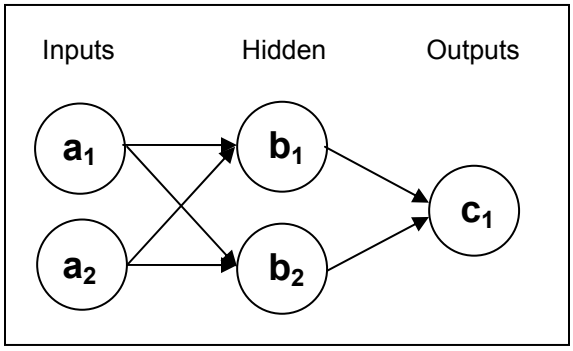


Figure 1: A Simple ANN

Each layer is connected to the next layer by a series of weighted connections which can have their values adjusted so as to produce different outputs. In most networks a bias input is added to the hidden layer and the output layer. This is simply implemented by treating the bias as an extra input in the input layer with a constant value of 1, such that the weight effectively becomes the bias. Similarly by adding an extra neuron to the hidden layer with a constant of 1 produces a bias input to the output layer. A bias value forces the network to produce a valued output even if the network receives 0 on all inputs. In the case of an artificial agent, if all sensors produced a 0 value, then the agent simply wouldn't move unless something passed in front of the sensors. Introducing these bias values to the input and hidden layers of the network means that the network will always (assuming the bias weight is non-zero) produce a non-zero output causing the agent to move.

Artificial neural networks lend themselves well to control systems as, apart from being biologically inspired, they conveniently convert a set of inputs to a set of outputs. In this case they can take a series of sensor inputs and convert them to motor outputs allowing the agent to move.

2.2 Recurrent Neural Networks

Recurrent neural networks are a slight variation on the traditional neural networks in that they contain connections within layers as well as connections across layers. The hidden layers of a network have connections between each of the neurons, as in Figure 2. This has the effect of giving the network a simple memory as each of the hidden nodes will have a state representing everything the network has seen so far. This "memory" means the network has the potential to produce a different set of outputs given the same set of inputs, which in some cases can prove beneficial. Recurrent neural networks are considerably more difficult to analyse as it is not always obvious what exactly is going on.

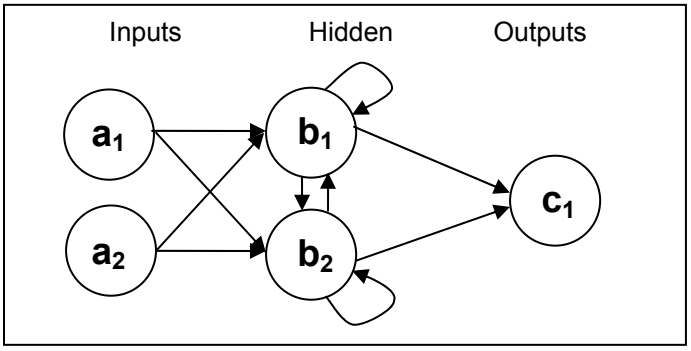


Figure 2: A Simple Recurrent Neural Network

2.3 Backpropagation

Backpropagation is a supervised learning technique. It requires a training set of input and output data which can be used to train the network. The process works by passing in each set of input values one at a time and comparing the output given by the network to the target outputs in the training data. This comparison will produce an error rate of how wrong each output was. The error is then propagated backwards through the network working out a delta weight change value for each weight in the network, according to a learning rate parameter, such that the error will be reduced.

The process of showing the network inputs is repeated many times until the error is suitably low. This is a very similar to a gradient descent method as it attempts to find weights that reduce the error. In many cases a small portion of the training data is held back for a cross-validation check to see how well the network performs on unseen data. This method is not part of the training process, but to see how well the network is currently performing. A network is not much good if it learns the training data so well that it cannot generalise to unseen cases. This is known as over-fitting.

Backpropagation is usually used to train networks for pattern finding or classification, where the network, when given some input information produces an output that attempts

to classify the data. In this project backpropagation will be used to train a neural controller by gathering training data from a hard-coded agent and see how well the trained neural controller can generalise to new sensory inputs when it is placed into the environment.

2.4 Genetic Algorithms

Genetic algorithms are another example of an artificial intelligence approach borrowed from nature. It is a form of search that is based on the process of natural evolution of a species. The genetic algorithm consists of a population of potential solutions each encoded by their own phenotype (characteristics) as specified by its genotype (genetic encoding).

The algorithm works by calculating a fitness score for each member of the population based on how good their solution is. When two members reproduce parts of both of them are combined (crossover) to produce an offspring which is a potentially new solution. The idea is that over time the fitter members of the population will get picked to reproduce, such that the offspring keep getting fitter and fitter. This is survival of the fittest, very much like it is in nature. For example, the animals that are better at avoiding the predators are more likely to survive and reproduce. Added to this method of crossover, each offspring has a chance of having some of its genetic information mutated. This is essentially a form of random change, which may or may not improve the fitness of the offspring. If crossover was the only technique used then eventually all possible available combinations will be used up, essentially halting the evolution process. The introduction of mutation can introduce new information allowing the evolution process to continue to find fitter solutions.

When using genetic algorithms to evolve neural networks the most obvious genotype setup is to represent each of the weights in the network, as changing these is was causes changes in the networks output. There are many variations of the genetic algorithm such as the microbial algorithm [5], tournament selection, roulette-wheel selection, and even some that do not use crossover at all. For the exact method that was used in this project see Section 3.

2.5 Subsumption Architecture

Subsumption architecture was first suggested by Brooks [2] as a way of creating a controller for a mobile robot. It is very difficult to produce a controller with many sensors that should accomplish multiple tasks so this technique introduces a way of breaking down the task into several smaller tasks. It involves breaking the desired task down into layers of simpler modules. The idea is that each layer deals with a single behaviour, with the most basic behaviour coming first. As the behaviours become more complex they build on each other, allowing these higher layers to incorporate and interefere with the lower layers (Figure 3). The lower layers will continue to work, but will have no knowledge that the higher layers even exist.

For example it would start off with a module for collision avoidance and once that has been sufficiently learned, a module would be added for wandering around the environment. Each higher layer that is added can subsume the layers beneath it, suppressing their outputs. This solves the complexity issue of building controllers as

each layer deals with an individual goal or sub-goal, and each layer only needs to have inputs from the sensors it requires in order to complete the goal.

Brooks uses of subsumption architecture did not involve the use of neural networks for each layer, however these can of course be used, and it is a useful way of simplifying the responsibilities of an individual network. Rather than using a single network with many inputs and a large number of neurons attempting to do everything, it is possible to create multiple smaller networks with fewer neurons, each with their own responsibilities. This means that there are fewer weighted connections required in the networks, thus speeding up computation time. The trick here is getting these layers ordered the correct way such that layers are subsumed correctly creating the desired outcome.

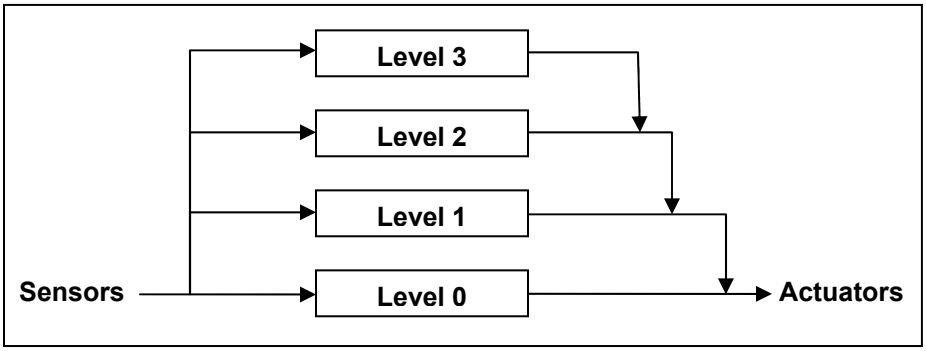


Figure 3: Subsumption Architecture, with higher layers subsuming the lower layers

3. Methods

It must be noted at this stage that all of this work was performed within a simulation developed specifically for this project. Section 4 will go into more detail about the actual design of this system. The simulation in no way attempts to model real world physics and has not been designed to allow for the resulting neural controllers to be transferred into physical robots. This is because the interest lies within how the learning techniques perform against each other and as such a relatively simple simulation environment is all that is needed. Evolving in simulation is also significantly faster than having to download each neural controller individually to a physical robot and then test it.

3.1 The Agents

Agents are represented in the simulation by a circle 25 pixels in diameter with a straight line showing the direction of the agent (Figure 4). Each agent has a left and right motor, which aren't visually represented, and a series of sensors. An agent can have two types of sensors, an obstacle sensor and a flag sensor. An obstacle sensor is essentially a proximity sensor and can detect obstacles such as walls and other agents. A flag sensor is essentially a light sensor that can be tuned to detect specific colours of light, in this case red or blue.

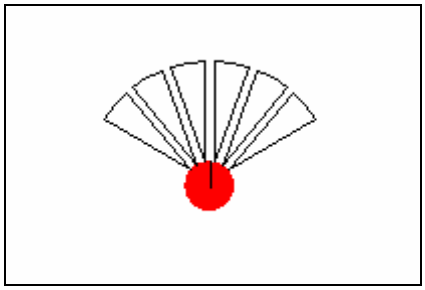


Figure 4: An agent with 6 sensors

Each sensor produces a value between 0 and 1, where 1 is close to the sensor and 0 is when nothing is detected. Sensors have three attributes that can be changed, offset, view angle and distance (Figure 5). Offset is the offset from the "nose" of the agent in degrees. View angle is angle width of the sensor in degrees. Distance is the maximum distance that the sensor can detect in pixels.

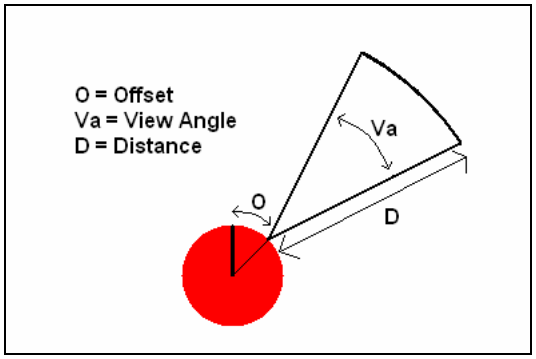


Figure 5: Sensor and it attributes

Sensors can be visible or invisible within the simulation. If they are visible they appear as in Figure 4, and if they are triggered (producing a value > 0) they turn blue.

Agents also have an additional boolean input to their controllers which is either 0, meaning they don't currently have the flag, or 1, meaning they do.

Agents also have a controller which is what converts the sensory inputs to motor outputs. For the agents being trained through backpropagation this contains a single neural network, where as for the agents being evolved by a genetic algorithm this can contain many layers of neural networks stacked on top of each other, with the higher layers subsuming the lower layers.

The motor speeds are values between -1 and 1, where a negative value means the motor is turning backwards. The speed of the agent is worked out by (leftMotor + rightMotor) x 2. So with both motors turning at full speed (value of 1) the speed of the agent would be (1+1) x 2 = 4 pixels/per time-step. Moving an agent at any angle in a 2-dimensional coordinate system with origin (0, 0) in the top left corner requires the following formulae.

$x = x + s * cos(a)$
$y = y – s * sin(a)$
where, x and y are the coordinate positions of the agent, s is its speed, and a is its current angle in radians. Angles in java work anticlockwise, with 0^o/radians situated horizontally to the right.

3.2 The Environment

The environment is a 2-dimensional world seen from above. The environment contains three main objects, the agents, the flags/bases and walls (Figure 6). Flags/bases are represented by coloured squares, 20 x 20 pixels, with a simple icon representing whether the flag is there or not. When an agent takes the flag this icon will appear "attached" to the agent as a visual representation that it is carrying the flag. If the flag has been taken by an agent then no other agent of the same team will be able to collect the flag. Once a flag has been taken and carried to the agents base then it will reappear at its original location ready to be taken again. Walls are black in colour and are either long and thin, such as the walls surrounding the environment, or are box shaped with size 40 x 40 pixels (representing an obstacle). When in tournament mode, when two teams are competing against each other, the environment is rotationally symmetrical. The reason behind this is to give each team an equal chance for comparison purposes, as it wouldn't be fair if one flag was easier to find than the other.

The environment contains simple collision detection. If an agent collides with a wall they will stop and their motors will stall for 20 time-steps. If two agents collide they will both bounce back a little, and again their motors will stall for 20 time-steps. Flag sensors cannot detect flags the other side of a wall. In other words if a wall intersects the line of sight between the sensor and the flag it will return a value of 0. Other agents, however, do not block a sensors view as it is assumed that a flag is taller than an agent such that the sensors can detect a flag over the top of an agent.

A team scores a point when an agent from that team successfully finds the opponent teams flag and carries the flag back to its own base. The winning team is the team that scores the most points within a given number of time-steps.

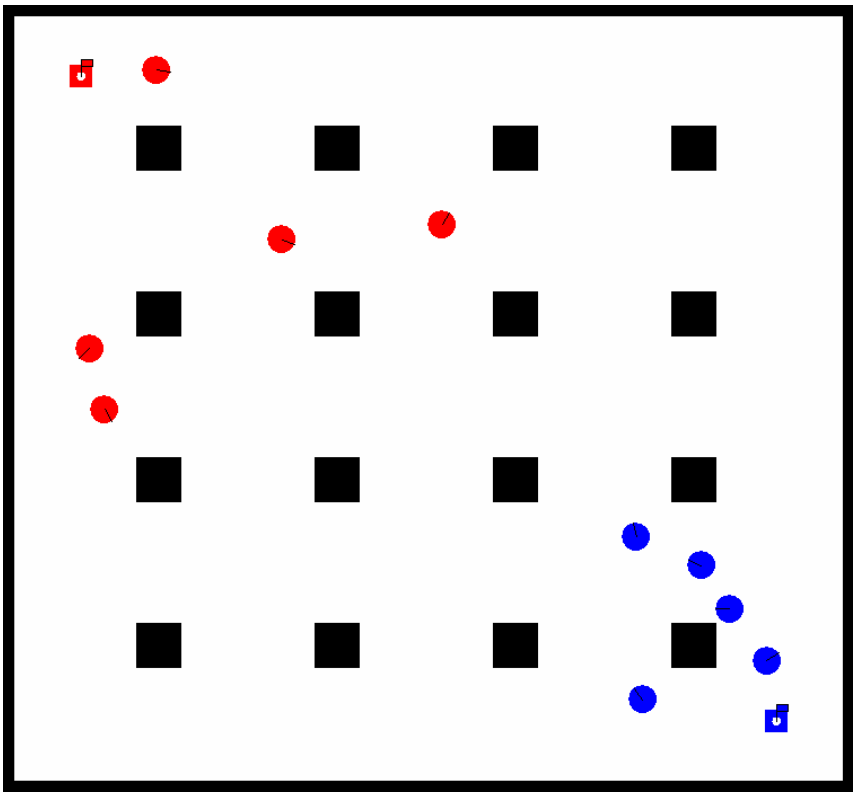


Figure 6: Environment, with 2 flags, walls and agents – note that sensors are not visible in this example

3.3 The Controllers

The control systems vary slightly depending on which learning technique is being used, in that the one being trained through backpropagation consists of a single network, whereas the one being evolved consists of several networks layered on top of each other. The network resulting from the backpropagation technique is a simple feed-forward neural network where at each time-step the sensory inputs are fed into the network producing the motor outputs.

The controller resulting from the genetic algorithm approach contains several layers of recurrent neural networks. A recurrent neural network is where each hidden node is connected to every other hidden node including itself. This is implemented using a copy hidden layer (Figure 7), where at each time-step the hidden layer values are calculated by the sum of all inputs plus the sum of the entire copy layer. The copy layer then is set to equal the new hidden layer values. This gives the network a short term memory and in the case of mobile robots can help to prevent them from getting stuck.

The controller contains several layers of these networks. The bottom (first) network has only 2 outputs corresponding to the left and right motor speeds. Any subsequent networks have 3 outputs, the first 2 being the motor speeds and the third is fed into a further black-box which controls the subsumption process (Figure 8). If this third value is

greater than 0.5 then this layers motor signal outputs are sent to the motors, unless a higher layer also has a value greater than 0.5, in which case that layers motor signal outputs are used. This is a similar technique to what was used in [11] and [12].

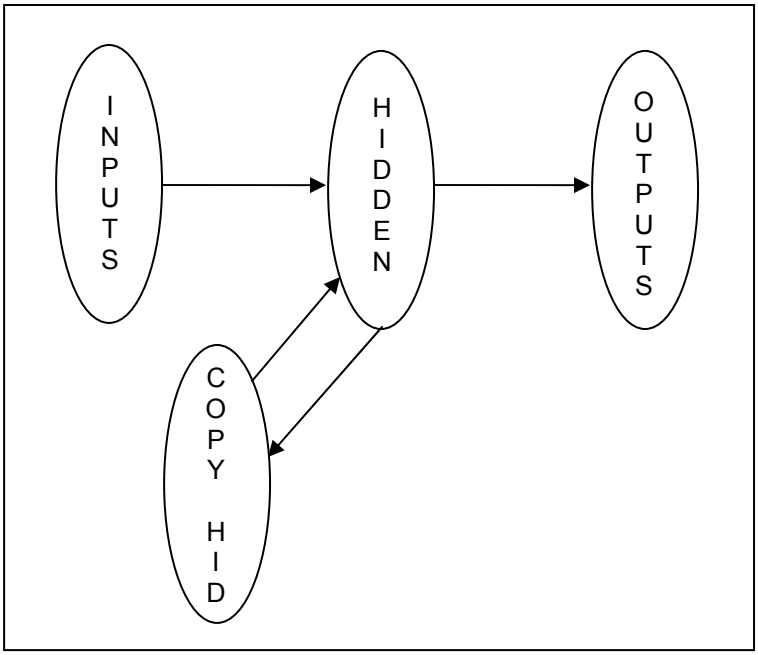


Figure 7: Implementation of a recurrent neural network

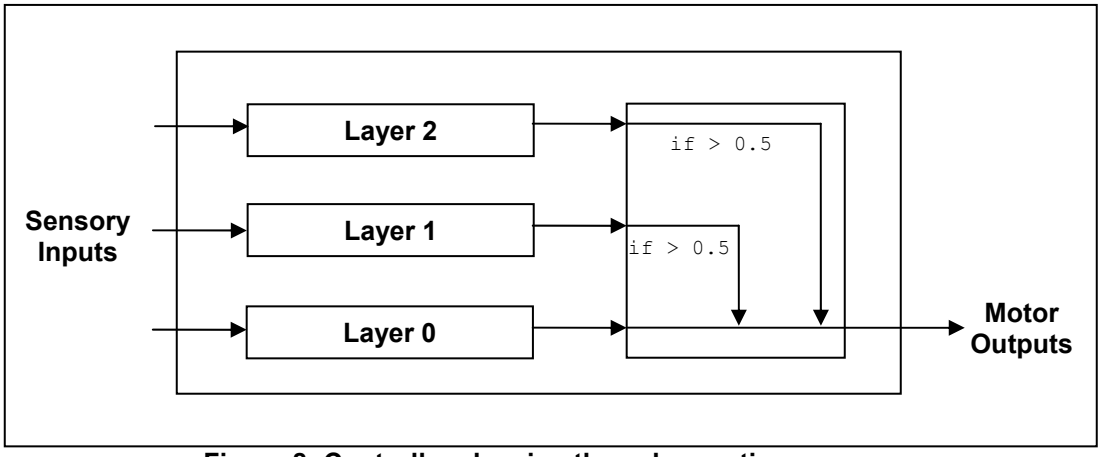


Figure 8: Controller showing the subsumption process

The activation function used within each node of the individual neural network is the tanh (hyperbolic tangent) function which squashes the output into the range -1 to 1. This allows the outputs to be fed directly to the motors producing the forward or backward speed. The output *y* of a node *j* in a network is calculated by:

$$y_j = \tanh\left[\sum_{i=1}^{n}(x_i w_{ij})\right]$$

where, x_i is an input value to the neuron, j, w_{ij} is the weighted connection from the input i to the neuron j, and n is the number of inputs to the neuron.

3.4 The Genetic Algorithm

The genetic algorithm works by evolving one layer at a time, and then freezing that layers configuration before adding the next layer on top. Therefore the genotype only encodes the weights of the top layer in the controller. The genotype is as follows:

[Input to Hidden weights] [Hidden to Output weights] [Copy Hidden to Hidden weights]

where all weights can have a value between -1 and 1. The exact algorithm used for creating the next generation is roulette-wheel selection with elitism added in. The idea of roulette-wheel selection is that each member is given a chance to be chosen as a parent in proportion to their fitness. This means that members with a higher fitness score have more chance of being chosen for reproducing; however it doesn't prevent less fit members being chosen. The advantage of this technique is that a phenotype with a low fitness may actually contain some useful information, that when combined with another produces a fitter member of the population. Elitism is also added which picks the fittest two members of each generation and automatically copies them into the next generation. This is done to prevent a loss of good solutions throughout generations.

Children are produced by randomly splitting both parents at the same point in their phenotype. Two children are then produced, where the first child is a combination of the first part of the first parent and the second part of the second parent, and vice-versa for the second child (Figure 9). Each gene in a child has a chance of being mutated whereby its value is randomly changed. This is repeated as many times as necessary taking a new random split each time, thus producing different children.

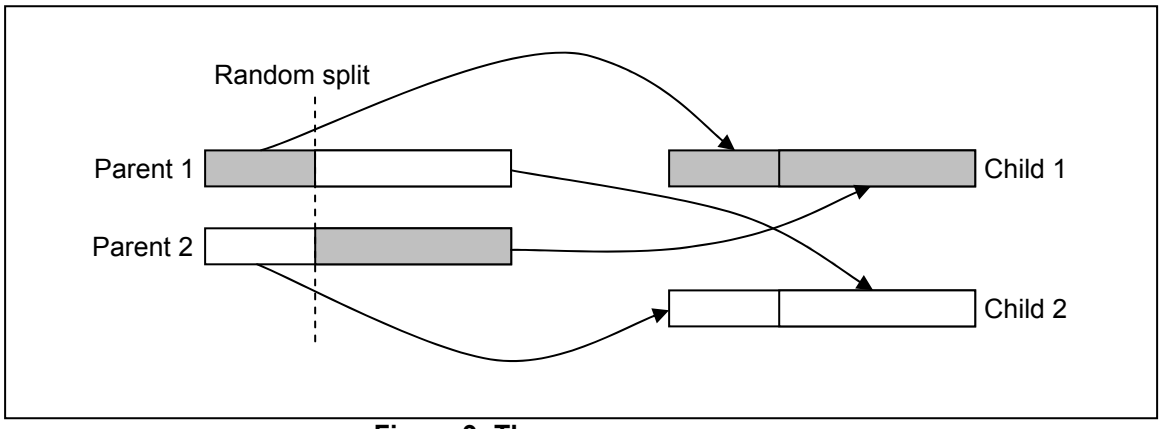


Figure 9: The crossover process

The process of producing a new generation is therefore as follows:

- The fittest two members are copied into the next generation (Elitism)
- Two other members are chosen as parents using roulette-wheel selection (Selection)

- Both parents are copied into the next generation, unless they happen to be either of the two fittest members, in which case they are not (as they are already there)
- All remaining members are replaced by the children of the parents (Crossover)
- Each gene in a child will have a chance of being mutated (Mutation).

3.5 The Backpropagation Approach

This approach requires some training data in order for the neural network to learn. This is achieved by creating an agent with 4 obstacle sensors and 4 flag sensors, where 2 of the flag sensors can detect the red flag and the other 2, the blue flag. This agent doesn't have a neural network inside, but rather a series of "if" statements, which based on the sensory inputs, produces different motor outputs.

This agent is then put into an environment with a randomly placed set of obstacles and two flags (one red, one blue) and allowed to move around for a number of time-steps. At each time-step the current sensor values and motor values are recorded producing the training data. A neural network is then created with the same number of input and output nodes, and a number of hidden nodes. The connection weights are randomly initialised to between -1 and 1 and the network is trained for a number of epochs. 10% of the training data is held back as test data to see how well the network can generalise to new inputs. This 10% is chosen at random from the training data. The training of the network is performed outside of the agent as it is only getting passed already known inputs and calculating the outputs. It is also faster if the network is trained outside of the agent, so it saves time.

4. Requirements Analysis

As this project is a research project their will not be many end users, however there is still a need to perform a full user requirements analysis for the interface. The main user will be the author as the developed simulation will be used to compare the two approaches mentioned earlier in the introduction. There is however a possibility that the system may be used by others for similar research, so the interface must be easy to use and intuitive.

4.1 Requirements Specification

Requirements Specification of the System -
"The system will be used as a simple research tool for comparing different techniques for producing desired behaviour in some artificial robots (agents) when playing a game of capture the flag. Each robot will have a series of sensors for detecting opponents, team-mates, flags and bases, as well as two motors attached to wheels (one on each side of the robot), similar to a Khepera robot or Braitenberg vehicle. Each robot will have a neural controller (Artificial Neural Network) which will be connected to the sensors and to the output motors. The user interface will have an arena in which the simulations can be run. The simulation will be a 2-dimensional birds-eye view of the world, and will not need to incorporate real-world physics as the main purpose of the system is for comparing the two training techniques. There will also be a series of options for editing the settings of the two learning techniques; back-propagation and the genetic algorithm. There must also be the ability to save the progress of both techniques so that the user can carry on from where the program left off last time."

Requirements of the Research -
"Use the developed system to compare the techniques of back-propagation and genetic algorithms when training a team of robots to compete in a simple game of capture the flag. One team will be trained using back-propagation and the other team will be evolved using a genetic algorithm. The genetic algorithm will have a population of genotypes with each gene representing a weight in the neural network. These will be evolved by testing each ANN in the arena and comparing which one performs the best. Whichever one performs the best will be evolved to produce new offspring. The back-propagation method will be a form of hard-coding as it is required that the expected outputs for a particular set of inputs be known. This will involve deciding what the robot should do when it receives certain inputs. Both techniques will be compared by pitting the teams against each other using the best controller from each technique to see how they perform against each other. The results of these experiments will then be graphed and analysed to see how they change over time."

4.2 Determining Classes and their Relationships

On performing a grammatical parse of the requirements specification in section 3.1 and a little bit of common sense we can determine the following classes:

- Robot
- Sensor
- Motor
- Flag
- Base
- ANN
- UserInterface
- Arena
- BackPropagation
- GA

If the classes are then grouped together according to their behaviours and how they interact with the rest of the system then it produces three main groups of classes. The first is the robots architecture, made up of the Robot, Sensor, Motor and ANN classes. The second group is the classes for the interface; the UserInterface, Arena, Base and Flag classes. The last group is the algorithms themselves; the BackPropagation and GA classes. Although these two classes are only loosely related they will be working on the same classes and so have therefore been put together.

CRC (Class-Responsibility-Collaborator) cards were produced (see Appendix C) to find relationships between the above mentioned classes. When this was performed it was decided that the Motor class was not needed. This was because the Robot class was in charge of updating its position based on both motors speeds. The outputs of the ANN would be the motor speeds, so the Motor class would simply be storing a single speed variable. It is easier to implement this within the Robot class itself, so then the ANN simply passes the output variables to the Robot class which then uses them to update its position on the screen.

The CRC cards do not show all of the responsibilities of each class as these will be further improved on throughout the Design phase of the project, they are simply there as a guide to give a basic overview of what the system will be structured like. Figure 10 is a class diagram showing the classes and their relationships between one another. As can be seen there is a two directed association between the Robot class and the Sensor class. This is because although the Sensor class doesn't need to explicitly know about the Robot to which it is attached, it does need to know about other Robot objects in the arena in order to detect them, and to generate a sensor value.

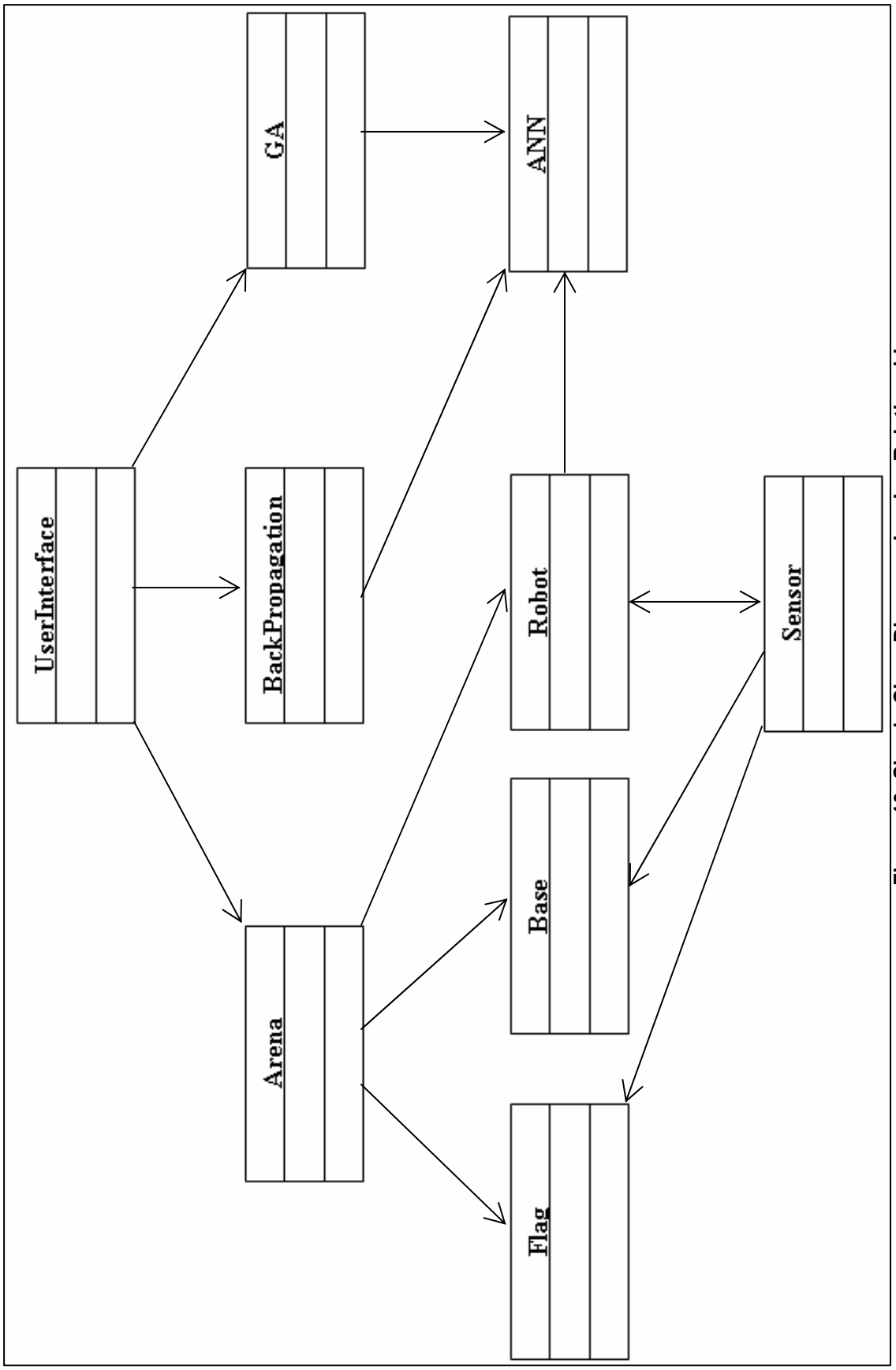


Figure 10: Simple Class Diagram showing Relationships

4.3 Implementation

Figure 11 (on the following page) is a more detailed class diagram including some of the basic fields and methods of each class. Several things must be mentioned here to help explain some of the decisions made about the fields and methods. Firstly the ANN class contains several fields which are double arrays. The first 3 arrays will hold the values of the input, hidden and output layers respectively. The size of these arrays would be determined upon construction of an ANN object. The bias values which will be added at the input layer and hidden layer will simply be treated as an extra input or hidden node with a constant value of 1 and will be added into the arrays upon creation of the object. The other two arrays are for holding the weight values for the connections between the layers and these will be the values that can be changed by either of the two training methods. Although a method has been mentioned in this class for calculating the sigmoid of a value, it may be decided during the research process that a different function such as tanh of a value is required instead to help towards the desired output.

Both the GA class and the BackPropagation class have several fields which allow for the different variables in the algorithms. For example in the GA class there is a field for the mutation rate, which will be able to be altered, so changing the behaviour of the algorithm to produce different (and hopefully better) results. The GA class has a fitness method which will be used to calculate the fitness of a particular genotype so that they can be compared easily. It also has an evolve method which will be used to produce new genotypes based on the crossover rate and the mutation rate as well as the type of evolution chosen. The BackPropagation class has two fields which are double arrays. These are to hold the delta error values for the hidden and output layers, which will then be used to adjust the weight values of the network in the right direction so as to improve the output it gives.

The Robot class will hold an array of Sensor objects allowing it to have as many as the designer requires. It also has two fields for the output (motor) values and a field for the angle (facing direction, in degrees) of the Robot. It is worth mentioning that the x and y fields are set as doubles simply because when calculating the updated position of the robot based on its speed and current direction it will involve working with decimal values, however when it comes to drawing the Robot on the screen these values will be cast to integers as obviously you can't have portions of pixels. Similarly the Sensor class has its x and y coordinates as doubles so that this gives a more precise position on the robot body. The offset variable holds the angle (in degrees) of the position of the sensor from the nose of the robot. The angle field holds the viewing angle of the sensor and the length field holds the maximum distance the sensor can detect something. Finally the value field holds the current input value of the sensor.

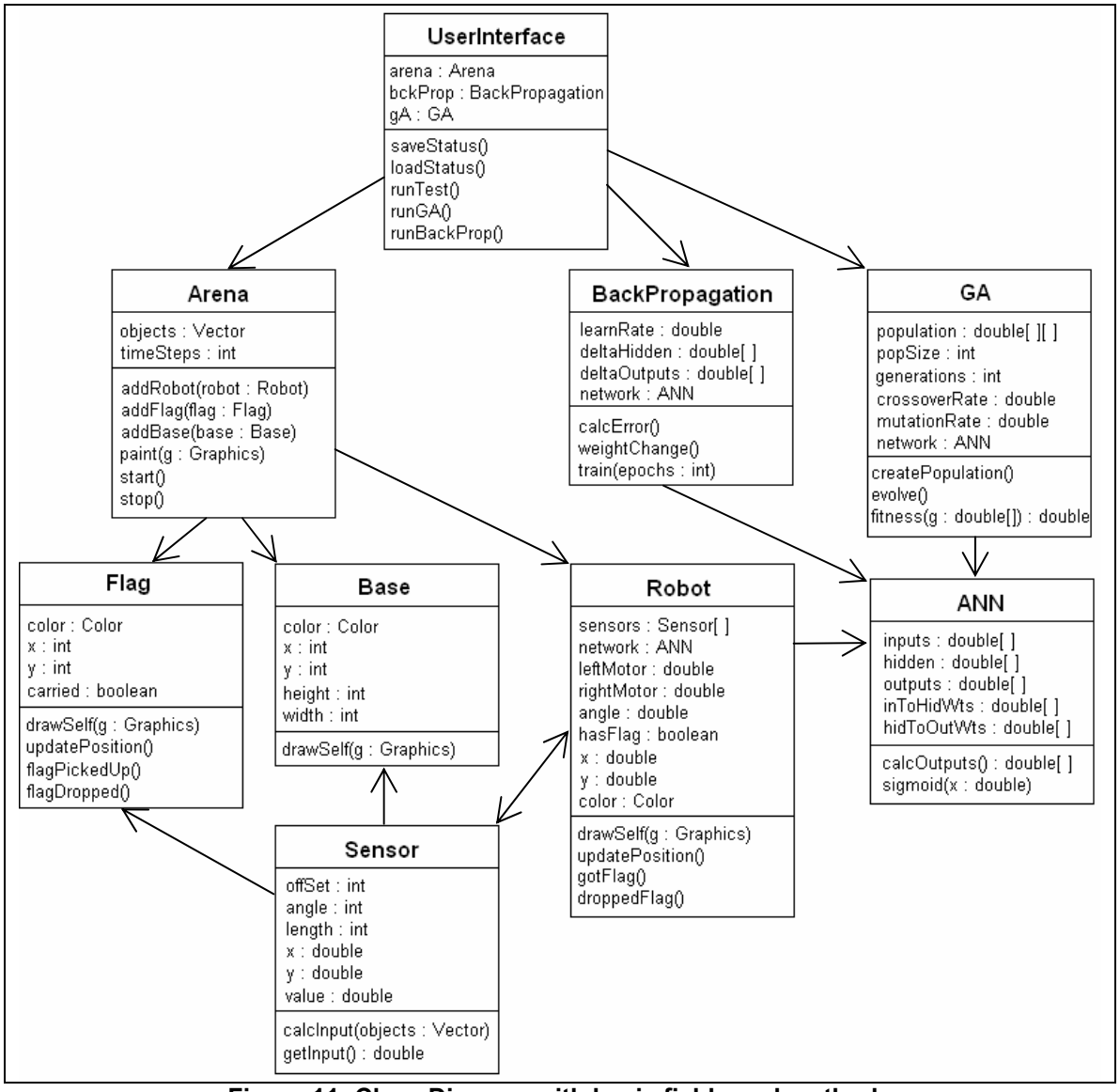


Figure 11: Class Diagram with basic fields and methods

The Flag and Base classes are very similar in that they have a colour field and an x and y position. The main difference is that the Flag class has methods for determining if it is currently being carried or not, and one for updating its position if it gets dropped by a robot.

The Arena class has a Vector which will hold all objects which will be in the arena, such as flags, robots and bases. It is clear here that a super class like ArenaObject will be necessary with classes Flag, Base and Robot extending this class but this will be implemented in the Design stage. The paint method will call each object in the Vector in turn and call its drawSelf method to redraw it in the correct position on the screen. It also has a start and stop method for controlling the simulation. The UserInterface class has several methods for loading and saving the status of the training processes and also for running the simulation and each of the training processes.

5. Design

Figure 12 on the following page shows a much more detailed class diagram. A dedicated loading and saving class has been created which takes some of the tasks away from the UserInterface class. The reason for the double directed link between the UserInterface and the LoadSave class is that the UserInterface needs to know about the LoadSave class in order to call its methods, and the LoadSave class needs to have access to the UserInterface to save the correct options that are displayed on the user interface.

The Arena class extends a Canvas and so has additional variables for holding a BufferedImage and its Graphics object. This allows the draw methods to draw everything to the image buffer first using its Graphics object, and then calling a single draw method to draw this buffered image onto the Arena. This technique is known as double buffering and prevents a flickering image as the entire screen is redrawn at once, rather than individual objects.

There is now an abstract superclass ArenaObject which is extended by the Flag, ArenaWall and Robot classes. The main reason for this is it simplifies adding objects to the arena. This class contains two abstract methods, drawSelf() and boundBox(). The drawSelf() method specifies how an object appears on the screen and will be different for each class. The boundBox() method creates a simple bounding box around the object which is then used for collision detection. If two bounding boxes intersect then a collision has occurred and the colliding objects should be moved apart.

The Flag and Base classes have been combined into a single Flag class. This decision came after noticing that the two classes were very similar. If a flag is dropped it is automatically returned to the base, so the variable for whether the flag is being carried or not is irrelevant. Instead the single class will either draw the flag as present at the base or not depending on whether the flag has been taken or not.

A Robot object now does not hold an ANN object but rather a Controller object, which in turn can hold many ANN objects. This creates a more generalised approach, whereby when the backpropagation technique is being used the Controller object will hold a single ANN object, but when the GA technique is being used it can hold more. Its calcOutputs() method will work out the output of each network it contains and determine which set of outputs will be returned to the Robot object for use as its motor speeds. This will be determined according to the process mentioned in section 3.3. The ANN class has been extended to include a copy hidden layer array as well as the associated weight array connecting the copy hidden layer to the hidden layer. There is also a boolean recurrent value which if true, will include the copy hidden layer in the calculation of the output, but ignore it otherwise.

A Fitness class has been added into the design to separate this process from the GA class. This class contains methods which for calculating the fitness of the passed in Robots controller. It is called by the GA class and returns the fitness for that particular controller. There is also a Graph class which is used by both the GA and BackPropagation classes. This class will be passed some data, and plot them to the screen. It will also scale the axes appropriately with the drawAxes() method and also add a legend with the drawLegend() method. The data it is passed will depend on what

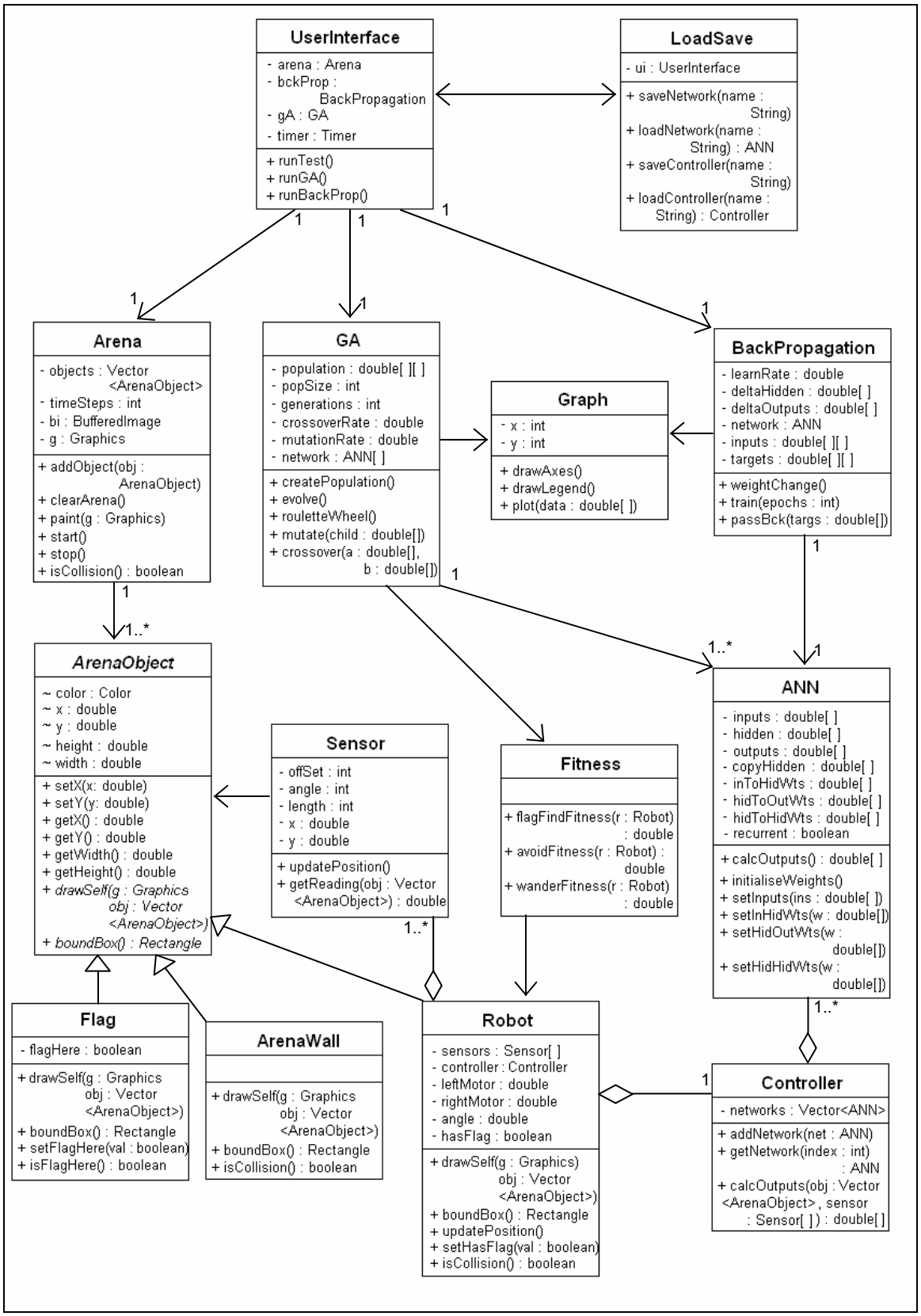


Figure 12: Detailed Class Diagram

class called it. If it is called by the GA class then it will get passed the average and best fitness scores across all generations. If it is called by the BackPropagation class then it will get passed the training and generalisation error over the set number of epochs.

The GA class has had several methods added to it to separate the process of choosing the parents, crossing over the genes of two parents and mutating a phenotype.

The UserInterface class is still simplified in Figure 12. The class will contain many java swing components, so listing all of these in the class diagram would not be particularly useful. It will also have several inner classes extending classes such as ActionListener attached to the buttons which will trigger events when these buttons are clicked. It will also have an ActionListener attached to the Timer object, which will call the Arena objects repaint() method every time the Timer fires.

5.1 GUI Design

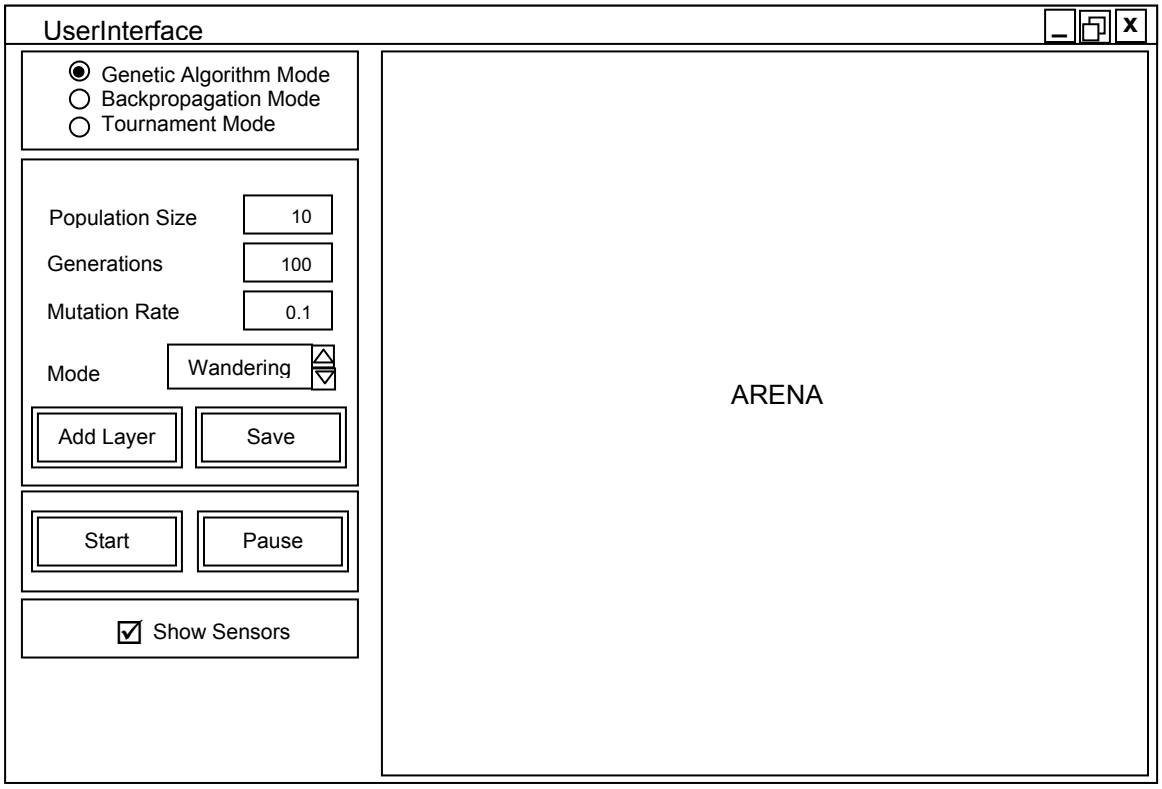


Figure 13: GUI Design

Figure 13 shows the basic design for the user interface. The main area on the right is an Arena object, which is where the simulation will be displayed. The top left hand box allows the user to select which mode they wish to use by highlighting the relevant option button. The panel beneath that will change depending which mode has been selected. Currently the GA mode is selected and so this panel displays options relevant to the genetic algorithm, such as population size, generations and mutation rate. It also has a mode option which will change what is actually drawn in the arena depending on what behaviour the user currently wishes to evolve.

The other panels that can replace the GA options panel are shown in Figure 14.

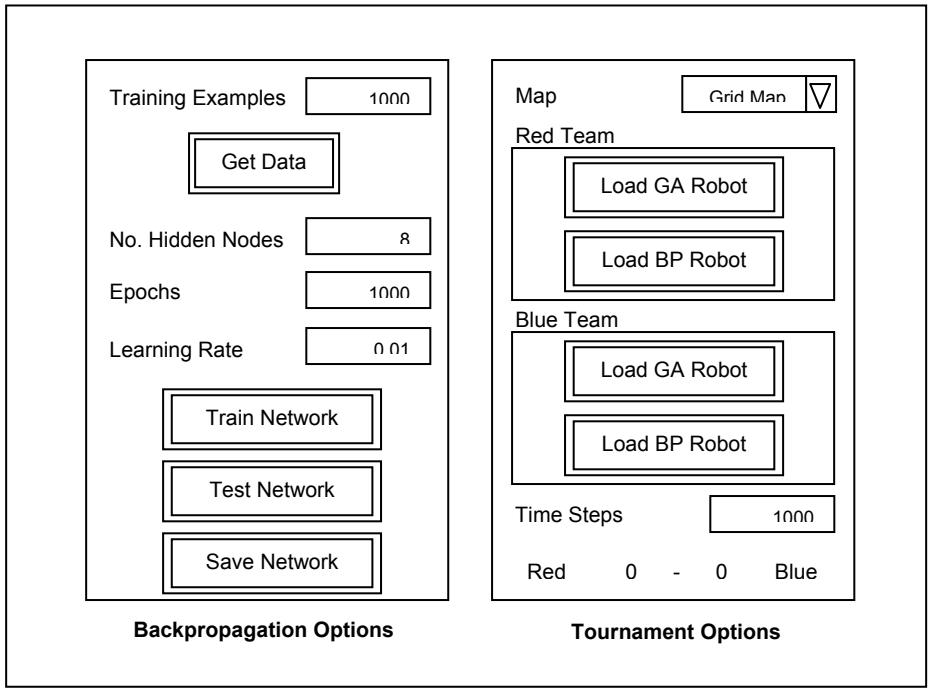


Figure 14: The alternative option windows for different modes.

The backpropagation options window has a button at the top which starts the hard-coded agent moving around in a randomly generated arena. It will run until it has gathered enough training examples, as set by the user. The user can then specify the exact settings for each of the variables before training the network using the gathered training data. There is also a Test Network button which will place the trained network into a Robot in the arena so the user can see how well it actually performs.

The tournament options window allows the map to be chosen, and a load option to load in the relevant controller. It has a box for changing how long the tournament will run for as well as a score board to keep track of what the score is.

5.2 The Simulation

A key part of this program is how the simulation actually works. When the user clicks the Start button (or the Get Data or Test Network button in the case of the backpropagation panel) the timer object held in the UserInterface starts. This timer has an ActionListener attached to it which calls the Arenas repaint() method every time the timer fires. The Arenas paint() method loops through its objects Vector and calls the drawSelf() method on each object. Figure 15 shows the sequence diagram for updating the position of a Robot over a single time-step.

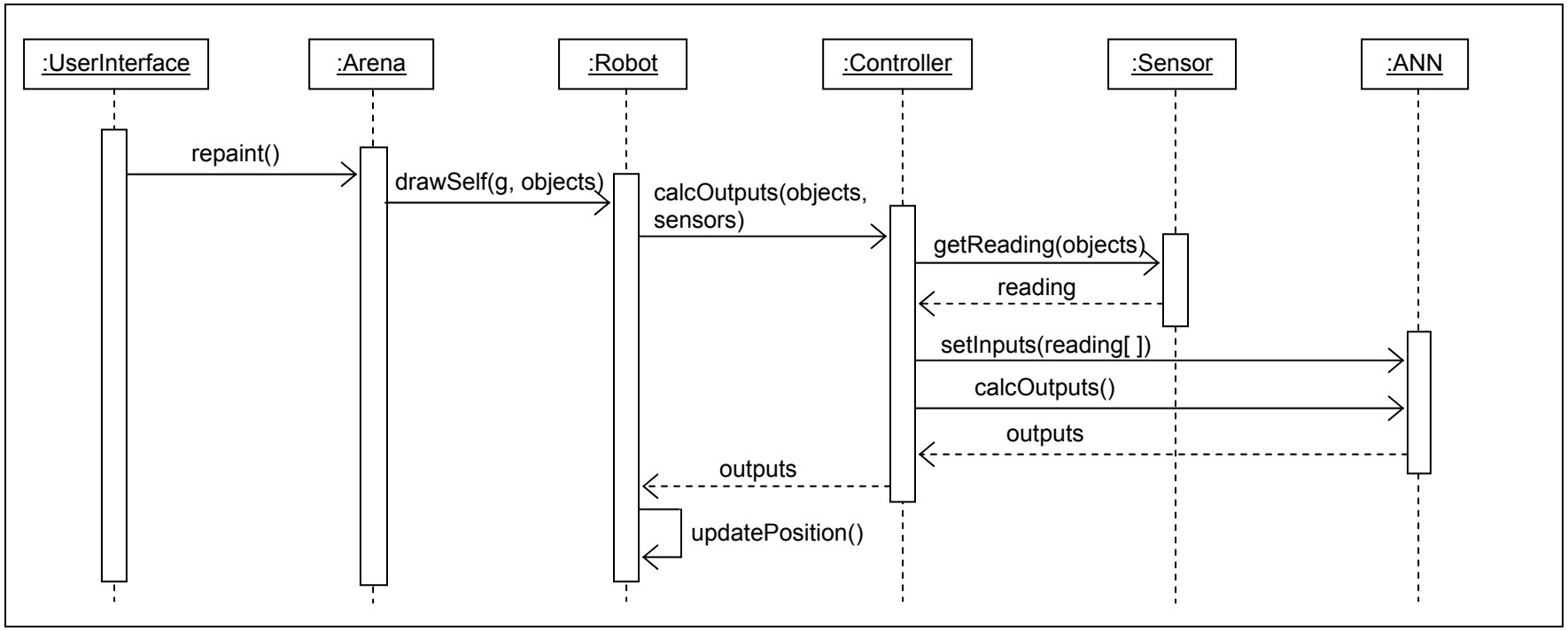


Figure 15: Sequence Diagram for a Robot moving position

The drawSelf() method gets passed the Graphics object of the Arena so that it can draw itself onto the Arenas buffered image. It also gets passed the entire Vector of objects held in the Arena. If the object being drawn is a Robot then the Vector gets passed into the getReading() method for each of the sensors. The getReading() method looks through this Vector and finds which objects are potentially within sensor range. It then attempts to see which of these is closest, and is actually within the detectable range of the sensor. Finally it works out how far away this object is and returns a distance value scaled between 0 and 1.

6. Evolution Experiments

The first task was to evolve a neural controller to play the game of capture the flag using the genetic algorithm approach. Throughout the evolution process each member of the current population was tested 4 times in different starting locations in order to get an average performance (fitness) score. The reason behind this is that on some occasions a certain situation may cause a different behaviour to occur thus improving or worsening the fitness function. Taking an average takes this into consideration.

The program works by allowing every member of the population to be tested at once, including all four of their runs. This means that if the population size is 10 then there will be 40 robots visible on the screen starting in 4 different places (Figure 16). The reason for this was that it significantly sped up the evolution process as rather than test every member of the population individually over four tests, all these tests were performed at once. Throughout the evolution process the agents were unable to see each other, as they were essentially being tested in separate identical environments.

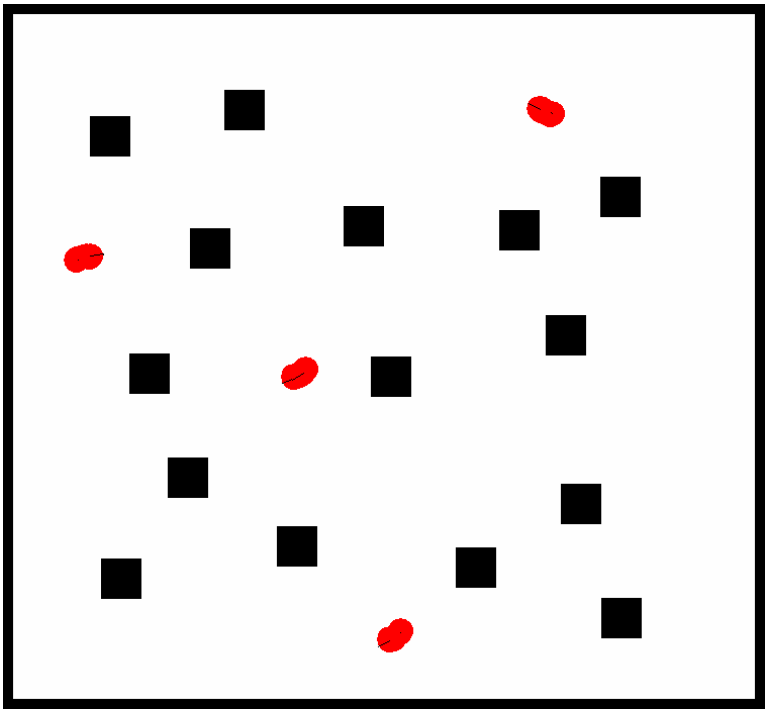


Figure 16: The start of a generation. Each of the 4 groups represent each of the 4 tests runs, with each group containing the same 10 members of the population.

Each generation was tested in the environment for 500 time-steps. The behaviours that can be evolved in the genetic algorithm mode of the program are wandering, collision avoidance and flag retrieving.

Attempting to evolve a wandering behaviour using a neural network is actually quite difficult. The simplest wandering behaviour is to encourage the agent to cover the largest distance, however this simply causes them to evolve to run in a straight line very fast, which isn't really "wandering" as you would expect. Neural networks by their very nature are input – output computational machines. If the agent only has sensors for

detecting obstacle and flags then in an empty environment these will always return 0. If the network keeps receiving the same input it will continue to produce the same output, so any more complex wandering behaviour would require additional inputs that are changing in order to produce a good wandering behaviour. This turned out not to be a problem in the end as when an agent is evolved to avoid collisions a natural wandering behaviour occurs as they change direction when they see an obstacle, such that over time they will have covered most of the arena.

6.1 Collision Avoiding Experiments

Collision avoiding was evolved by placing the agents in an environment with up to 20 randomly placed obstacles[2] (as in Figure 16). For each generation these objects were randomly moved to new positions and the agents starting locations and angles changed. This was to prevent the evolutionary process exploiting the environment to achieve a higher fitness. If the environment and their starting positions remained the same then it may be possible for them to evolve a simple behaviour, such as where they only turn left because from their starting positions they never encounter anything where they need to turn right to avoid it. This is not a good solution as when they are placed in an arena that does have obstacles they need to avoid by turning right they will fail to do so.

The fitness function for collision avoiding was:

$f = t \times a$

Where *t* was the number of time-steps where the agent wasn't colliding with anything, and *a* was the number of places visited. The maximum *t* could be was 500, as there were 500 time-steps per test, and was calculated by adding 1 for every time-step the agent successfully avoided hitting anything. Calculating the number of places visited involved splitting the arena up into a grid of smaller squares, 25 x 25 pixels. The value *a* was increased every time the agent moved into a grid square that it had not visited before. This helped to prevent a circling behaviour evolving; as once they had visited a location they couldn't increase their fitness by visiting it again. These values were multiplied together to force both to have an affect on the fitness. If the values were summed together then it could have been possible for an agent to evolve to gain fitness from one method but not the other. For example an agent could have just sat still as they would have still gained fitness for not hitting anything, or they may have moved incredibly fast and covered a large distance but crashed into every obstacle in sight, both of which aren't particularly good behaviours.

For each of these experiments in attempting to obtain the best collision avoiding behaviour the population size was 10. The number of hidden nodes tried was varied from 4 up to 10 in increments of 2. There were 6 obstacle input sensors in this first set of tests and the layout was the same as in Figure 4. Each test was run for 100 generations. Figure 17 on the following page shows the results of collision avoiding with 4 hidden nodes. It must be noted that although the program was capable of producing fitness and error graphs, all graphs in this section were produced in Matlab with the data gathered by the program. This is just because they are slightly clearer.

[2] The system will attempt to place 20 obstacles, but it cannot place two obstacles too close together, so if it cannot find a safe place to position an obstacle after 20 attempts, it will stop. Therefore the arena will not always contain 20 obstacles.

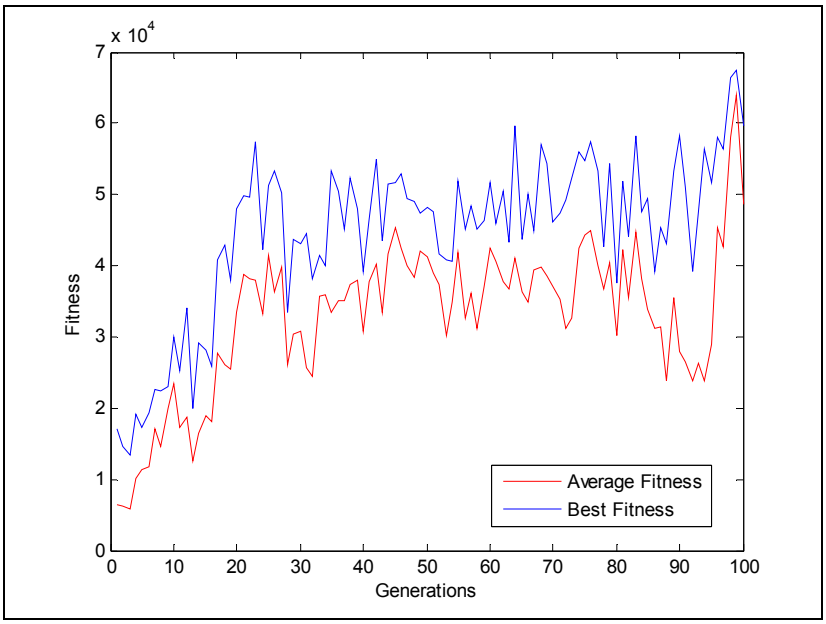


Figure 17: Fitness graph for collision avoiding with 4 hidden nodes

The graph shows that in the first 20 generations the fitness increased fairly rapidly, and then it started to level off. At about 80 generations there was a drop in the average fitness, possibly where at least one of the parents picked by the roulette wheel selection wasn't particularly good, and so the fitness dropped. Then in the last 5 generations a rapid improvement in fitness occured. There are several possible reasons for this. One reason is that the fittest two members which were passed over from previous generations became the parents and produced an overall fitter population. The other reason is that one of the children produced when the fitness dropped became a parent for the next generation and actually had some useful genetic material which caused a sudden increase in fitness. Another reason is that a mutation occurred which improved the fitness of a particular phenotype and then that one was chosen to reproduce.

Figure 18 shows the results of evolution process when using hidden nodes from 4 to 10. The graph shows that as the number of hidden nodes increased the fitness reached by the 100th generation decreased. The best fitness decreased at a fairly gentle rate whereas the average fitness of the population decreased a lot more rapidly. This implies that having fewer hidden nodes enabled the population to evolve towards better solutions faster. Figure 19 shows the average fitness plots for each number of hidden nodes overlaid over the top of each other. It can clearly be seen that after 15 generations the version with 4 hidden nodes continued to evolve at a fairly rapid rate, where as the others started to slow down. The mutation rate used in these tests was set to 0.1, which meant that every gene in a genotype had a 10% chance of mutation. Higher mutation rates were also tried but the overall fitness they produced seemed to vary quite a lot and in most cases was worse than when using a rate of 0.1. A mutation rate of 0.1 tended to produce more consistent results over time.

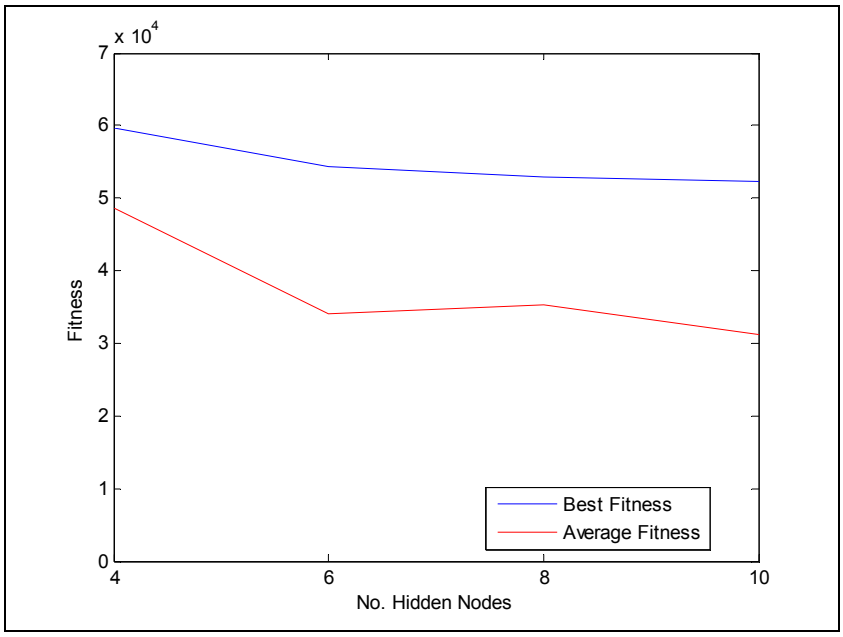


Figure 18: Graph showing fitness reached over 100 generations with varying hidden nodes

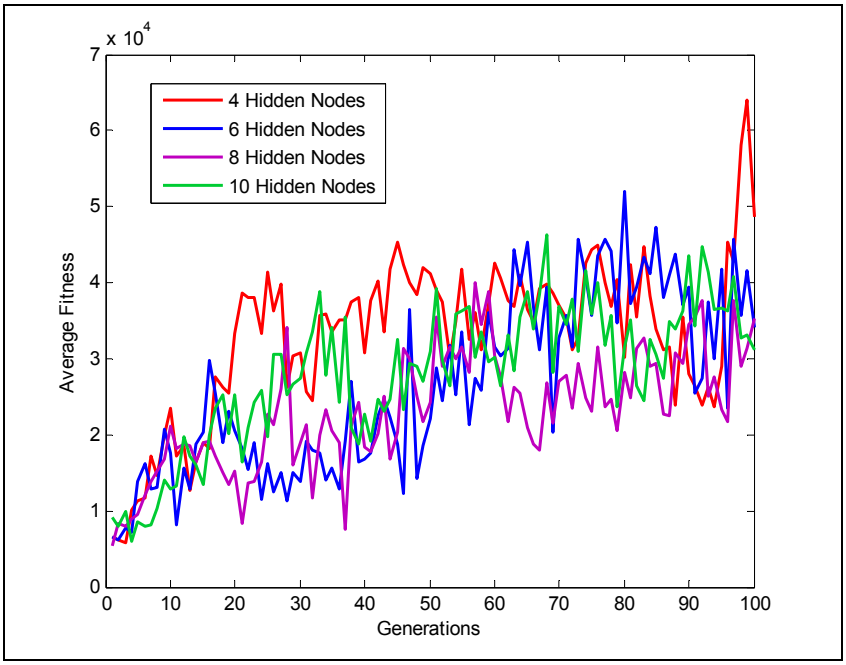


Figure 19: Average Fitness for varying numbers of hidden nodes

The collision avoidance layer of the neural controller was complete so the next stage was to add the next layer, the flag retrieval layer, on top. The architecture of the neural controller was basically that of a single multi-layer neural network, with 6 inputs, 4 hidden nodes and 2 output nodes.

6.2 Addition of Flag Retrieval Experiments

The addition of another layer to the controller was in itself straightforward. What wasn't straightforward, however, was selecting a suitable fitness function to encourage the controller to continue to use the previously evolved layer. Although the bottom layers weights had been fixed it was possible for the evolutionary process to evolve the top layer such that it always subsumed the bottom layer, meaning that although the bottom layer was working as it should, its outputs were never getting sent to the motors.

Choosing the correct fitness function was therefore critical in order to allow the balance to be maintained. In this case the idea was that when the agent cannot see the flag the bottom layer was to work as usual such that the agent moved around and avoided collisions. When the agent detected a flag, then the top layer should subsume the bottom layer and take control so as to steer the agent towards the flag.

The environment setup was similar to that for collision avoiding except for the addition of two flags, one red and one blue, placed at random in the arena (Figure 20). Again, the agents were placed in random starting locations each generation and the positions of the walls and flags also changed. The addition of this environmental noise was to try and avoid the genetic algorithm exploiting part of the environment.

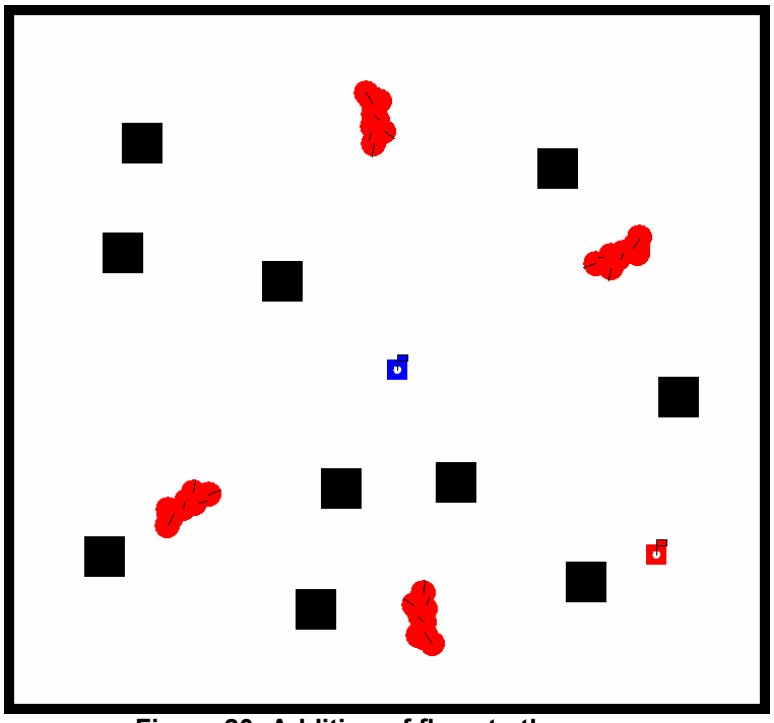


Figure 20: Addition of flags to the arena

The idea was that the controller should evolve to go to the blue flag first (collect it), and then it should find and go to the red flag to drop the blue flag off. Then it should repeat this process. It should be able to do this without crashing into anything.

The fitness function that was tested first was:

$f = p \times t$

where t was the time without a collision and p was the proximity to the flag. The bottom layer was the only one with obstacle sensors and such must be used if the agent was to successfully avoid hitting anything, so adding the value t encouraged the controller to continue to use the bottom layer. The value p was a cumulative value calculated as follows:

- For approaching the blue flag, if not already carrying a blue flag – give a value between 0 and 1, where 0 is not approaching, 0.5 is half way towards the flag, and 1 is right next to the flag.
- For hitting the blue flag, if not carrying a blue flag – add 12.
- For approaching the red flag, if carrying a blue flag – give a value between 0 and 1, same as above.
- For hitting the red flag, if carrying a blue flag – add 12.

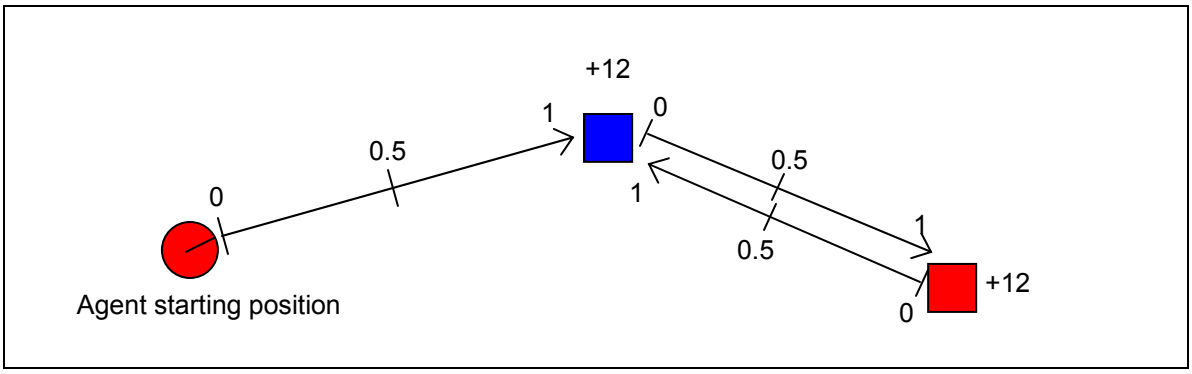


Figure 21: Proximity scoring system

So for an agent that had approached a blue flag, picked it up, and carried it half-way towards a red flag it would get a proximity fitness of 1 (for approaching blue flag) + 12 (for hitting blue flag) + 0.5 (for going half way towards red flag) = 13.5. An agent that moved towards, and hit the blue flag, but then never moved towards the red flag would end with a fitness of 13. An agent that never moved towards the blue flag would get a proximity fitness of 0.

The sensor setup for flag retrieval consisted of 4 sensors, 2 for detecting the opposing flag and 2 for detecting the same teams flag. 1 sensor for each flag was for sensing to the left of the agent and the other was for detecting to the right, as in Figure 22.

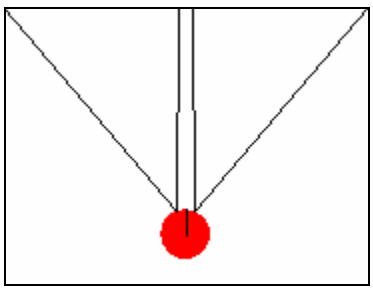


Figure 22: Flag sensor setup

The network for the flag retrieval layer had 5 inputs, where 4 were the flag sensors, as stated previously and the last was the boolean input as to whether the agent was currently carrying the flag or not. The test was run with 4 hidden nodes and for 100 generations with a mutation rate of 0.1 as before. Figure 23 shows the results of this test.

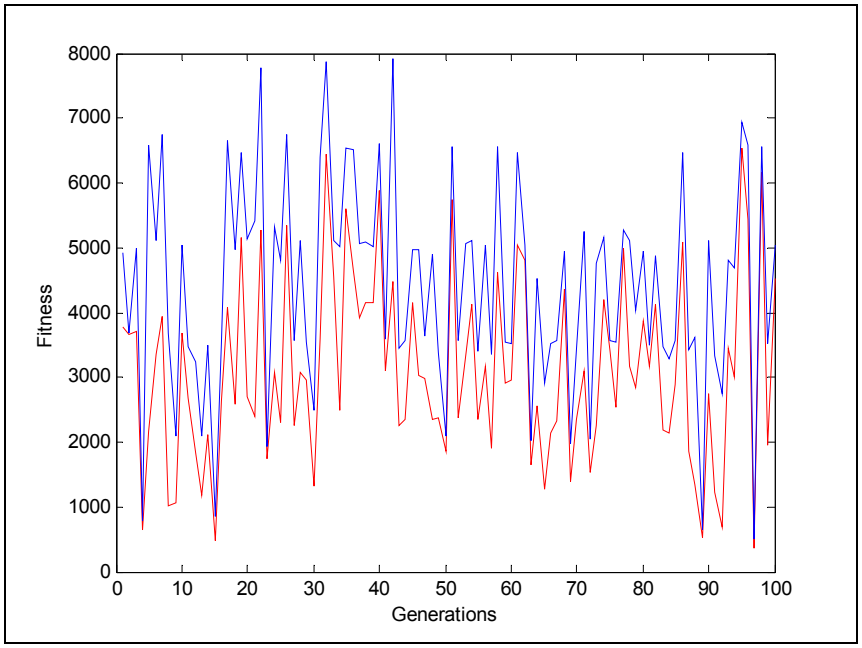


Figure 23: Fitness graph for flag retrieval with collision avoiding

As the graph shows the results aren't that good as the average fitness didn't appear to increase across 100 generations. The graph is very erratic where at one generation the average score is quite good, yet the next generation it suddenly drops. Although not clear from the graph, the solution half worked. What the resulting controller did was spin around on the spot unless it could see the blue flag, if it could then it moved towards the flag, avoiding walls en route and collected it. At this point the agent simply turned around on the spot over the blue flag, even if it could see the red flag.

The problem appeared to be that even using this fitness function which appeared to incorporate both layers of control the evolutionary algorithm struggled to settle down with one type of behaviour. If an agent had a controller which in the previous generation successfully found the flag, and then failed to find it in the current generation it wouldn't get a very good fitness, however it may have still been the best potential solution of the current generation even if the fitness function didn't confirm this. The genetic algorithm would not give it a good fitness and thus it got killed off and so the evolutionary advantage was lost.

Other fitness functions were attempted including just using the proximity value as the fitness of a particular controller, and also more complicated ones such as:

$f = p \times t \times a$

where p was the proximity value, t was the time without a collision, and a was the number of places visited, as before.

The idea with this fitness function was by adding a value for the number of places visited it might encourage the evolutionary process to prefer agents which moved around as well as find the flag and avoid collisions.

The results for just using the proximity value as the fitness are shown in Figure 24. As expected this doesn't cause any improvement as the graph clearly shows no learning across 100 generations. Using this fitness function caused the top layer to evolve such that the third output is always greater than 0.5, so the collision avoiding layer was never used. As a result, if the agent saw a flag and attempted to move towards it, then it would hit any walls that were in the way. As a result of the permanent subsuming of the collision avoidance layer the wandering behaviour was lost too, so the agent did not move around the environment unless it could actually see a flag.

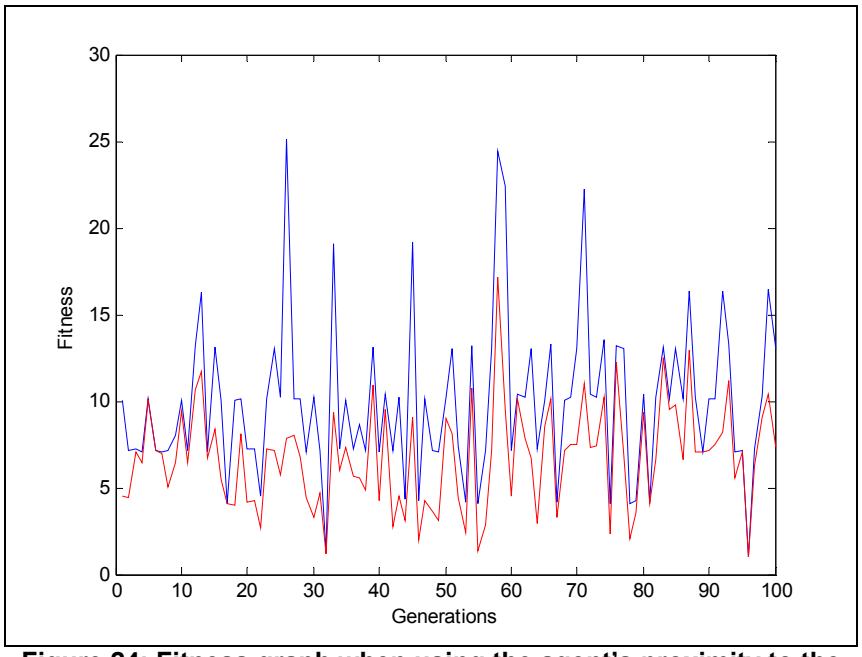


Figure 24: Fitness graph when using the agent's proximity to the flags as its fitness

Figure 25 shows the results of the more complex fitness function and as it shows the same problem has occurred where the graph is very erratic. It shows that it is important to pick a suitable fitness function; however it is also important to make sure the environment is setup correctly. The problem was that the environment was too noisy from one generation to the next so there was no consistency for the genetic algorithm to pick up on. Making the environment less noisy is difficult as if there is too little noise the genetic algorithm will exploit the environment and find an unexpected solution. Therefore the challenge is to find the balance between the environment complexity and the fitness function.

Even a relatively simple task like capture the flag is not that easy to evolve as it involves careful setup of the environment and the choosing of a suitable fitness function based on the environment. As the results so far have shown this is not a simple task.

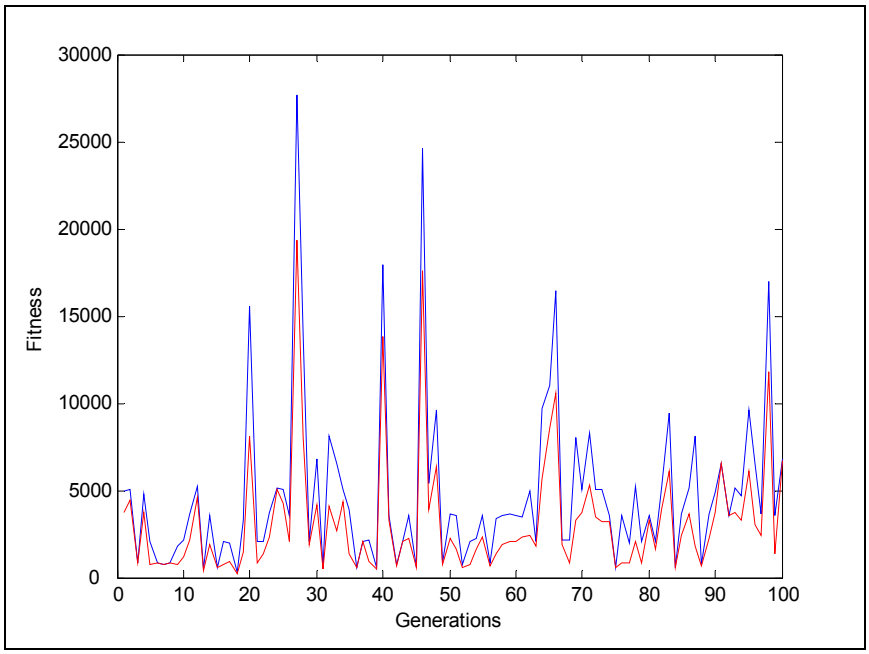


Figure 25: Fitness graph when using fitness function *f = p x t x a*

6.3 A Solution

The solution to this problem involved moving away from a purely genetic algorithm approach to a combination of a genetic algorithm with some engineering. The idea was to evolve both behaviours individually and then combine them together into layers after they have been evolved. This required some changes to how the subsumption process worked, moving away from the ideas in [11] and [12]. The solution is shown in Figure 26.

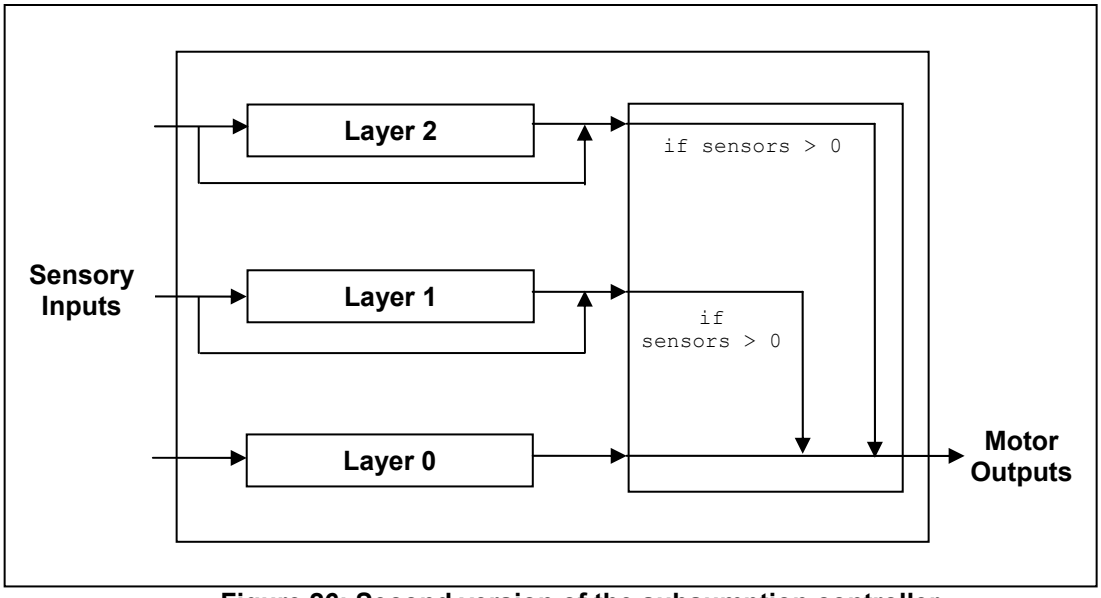


Figure 26: Second version of the subsumption controller

The solution involved passing the inputs to each layer into the black-box as well as the layers outputs. The check then involved no longer checking if the 3rd output was greater than 0.5, but to check that the layer was actually receiving an input from a sensor. This way the controller definitely uses the lower layers unless the higher layers are actually receiving a sensory input.

Due to this change, the flag retrieval layer could be trained in a simpler environment where there were 2 flags, and no walls. The blue flag remained in the centre of the screen from one generation to the next, while the starting positions of the agents and the red flag varied. The random positions of the agents could be anywhere on the perimeter of a circle with centre at the blue flag, so they all started at the same distance, however what was varied is the angle at which they were looking. The red flag was placed randomly on the screen at different positions, so as to vary the distance between itself and the blue flag. Adding these random position changes added some noise to the environment but not too much that the algorithm couldn't learn.

The fitness function could then be just the proximity value of the agent to the flag(s). The test was run with 4 hidden nodes, so the network architecture was 5 inputs (4 sensors + 1 boolean input), 4 hidden nodes and 2 outputs. The evolutionary process was run for 100 generations with a mutation rate of 0.1. Figure 27 shows the result.

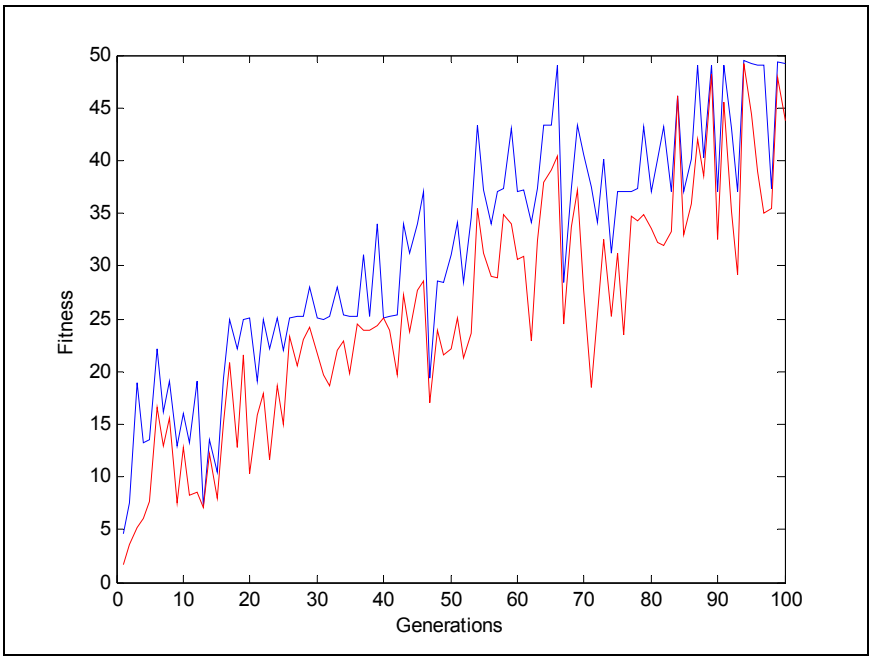


Figure 27: Fitness graph when evolving flag retrieval in a simpler environment

The graph shows a definite increase in fitness across the 100 generations at a constant rate until about 85 generations where it levels off. This is obviously a much better result than before, however this was when evolving the flag retrieval layer separately. After 5 generations the controller was able to reach the blue flag fairly consistently and by 30 generations it was able to take the blue flag to the red flag. The fitness increase after 30 generations was due to the controller learning to do this process faster and faster each time so by the 100th generation the controller was able to take the blue flag to the red flag and drop it off twice in 500 time-steps.

The test was also performed with varying numbers of hidden nodes from 4 – 10 with increments of 2. Figure 28 shows the results, and surprisingly the results are similar to those when evolving for collision avoidance, in that 4 hidden nodes appeared to be the best number. Using 6 hidden nodes seemed to produce a very erratic graph, with large dips in fitness every so often. When using 8 and 10 hidden nodes the controllers struggled to learn that once they had reached the blue flag, to then move on to the red flag as shown by the fairly flat sections of fitness. Using 10 hidden nodes caused a lack of learning beyond reaching the blue flag, as the average fitness never reached above 20, where a successful run from getting the blue flag to hitting the red flag would result in a fitness of over 20.

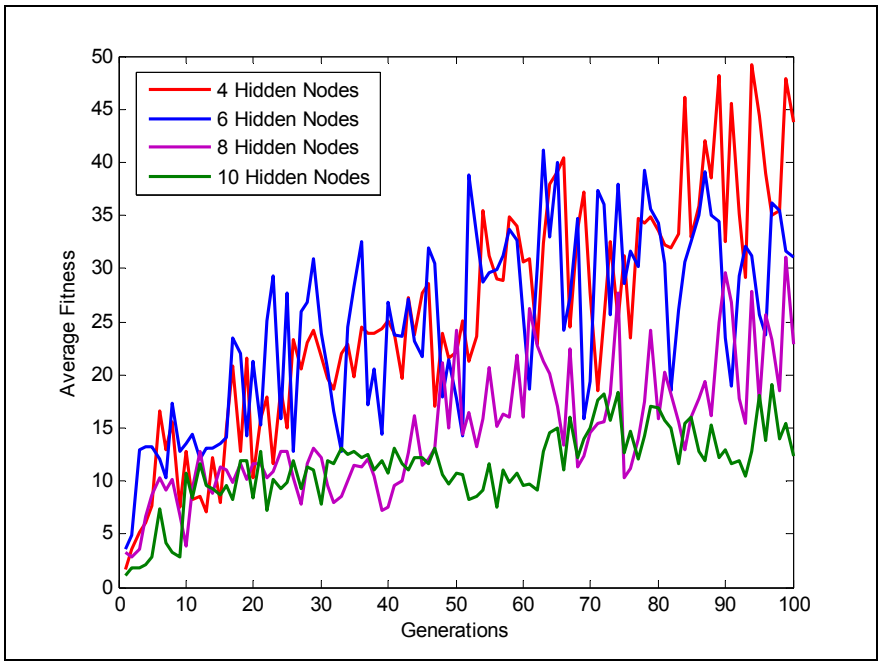


Figure 28: Average Fitness for different numbers of hidden nodes

7. Training Experiments

In order to use the backpropagation approach a set of training data was required. This was gathered by programming an agent to do the task of capture the flag and then capturing its sensor activations and motor outputs at each time-step across 5000 time-steps. This was then split into two groups, the first group comprising 90% of the captured data was used as training data and the final 10% was used as a generalisation set to check the performance of the network. The generalisation set was picked at random from the captured data set.

It turns out that programming an agent based on its sensor inputs is not an easy task as there is difficulty in determining what to do in the case of every possible set of sensory inputs. However an agent was produced that worked well enough to gather the 5000 sets of training data. In theory, the backpropagation learning technique should be able to generalise well enough to cope with the lack of precise control initially programmed into the original. The coded agent had 8 input sensors plus the boolean value for whether it had a flag or not. The 8 sensors include 4 flag sensors, 2 for each colour flag, and 4 obstacle sensors, as opposed to the 6 used in the genetic algorithm approach. The main reason for this was that it was easier to program the collision avoiding behaviour with fewer sensors, although still producing the desired effect. Figure 29 shows this sensor setup.

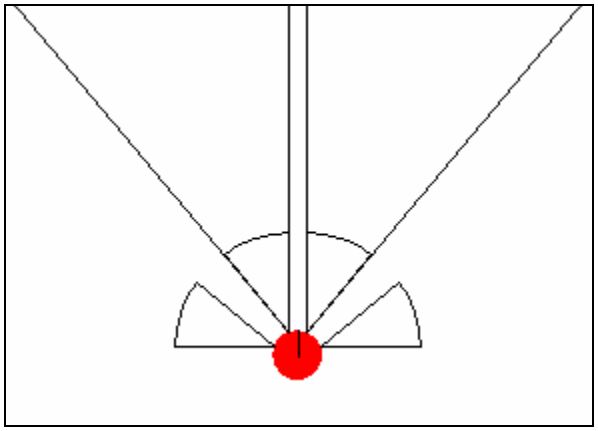


Figure 29: Coded robot sensor setup, the long sensors are flag sensors, and the short ones are obstacle sensors

The approach also used a small amount of weight decay on the hidden to output weights, to encourage the weights to remain small. Weight decay helps prevent the weights getting so big that the network starts to overfit the training data. Weight decay was implemented by multiplying the old weight value by a small constant before adding the weight change value. The small constant in this case was 0.9999, so the weights would shrink by only a tiny amount each time the weights were updated.

Varying numbers of hidden nodes were tested for the neural controller, from 2 – 14 in increments of 2. Figure 30 shows the generalisation performance results of these tests. The graph shows that generally the more hidden node there were the better the network performed. With only 2 hidden nodes the generalisation performance was particularly bad, and actually got worse after about 25 epochs.

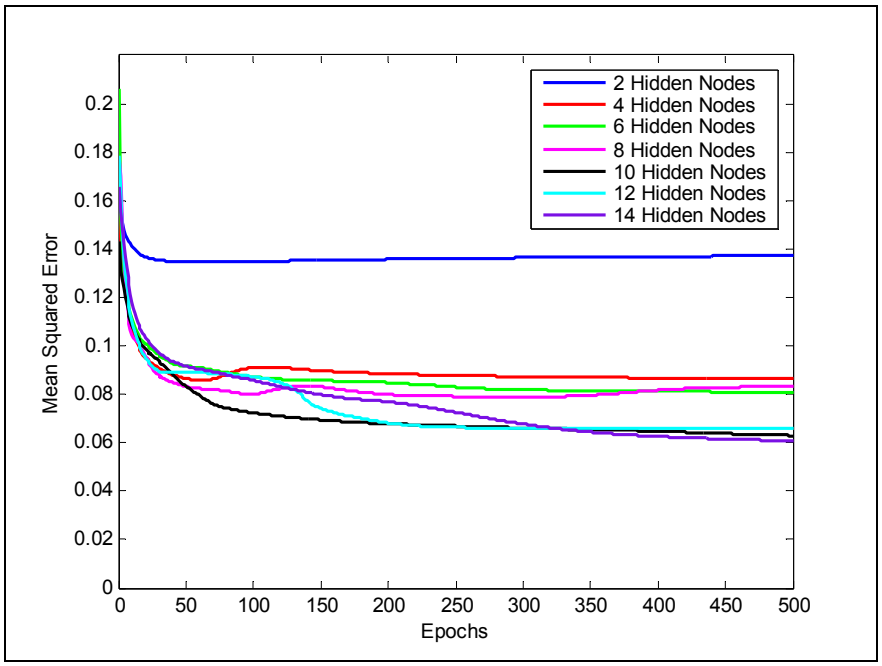


Figure 30: Generalisation error with different numbers of hidden nodes

Using 14 hidden nodes produced the best results over 500 epochs, however 10 or 12 also produced similarly good results. The above test was done with a learning rate (eta) of 0.1. Figure 31, on the following page, shows the results of different learning rates when using 14 hidden nodes. The results are less consistent than those from figure 30, although what can be seen is that with a low learning rate of 0.01 the generalisation error curve is very smooth, but it learns quite slowly. Using higher learning rates between 0.05 and 0.15 caused much quicker learning, where the minimum generalisation error reached with a learning rate of 0.01 in 500 epochs, was reached in 75 epochs. A learning rate of higher than 0.2 caused a more erratic generalisation curve, and higher than 0.25 produced a worse error than when using a low value of 0.01. A learning rate of 0.05 or 0.1 seemed to produce the minimum generalisation error and produce the best performance.

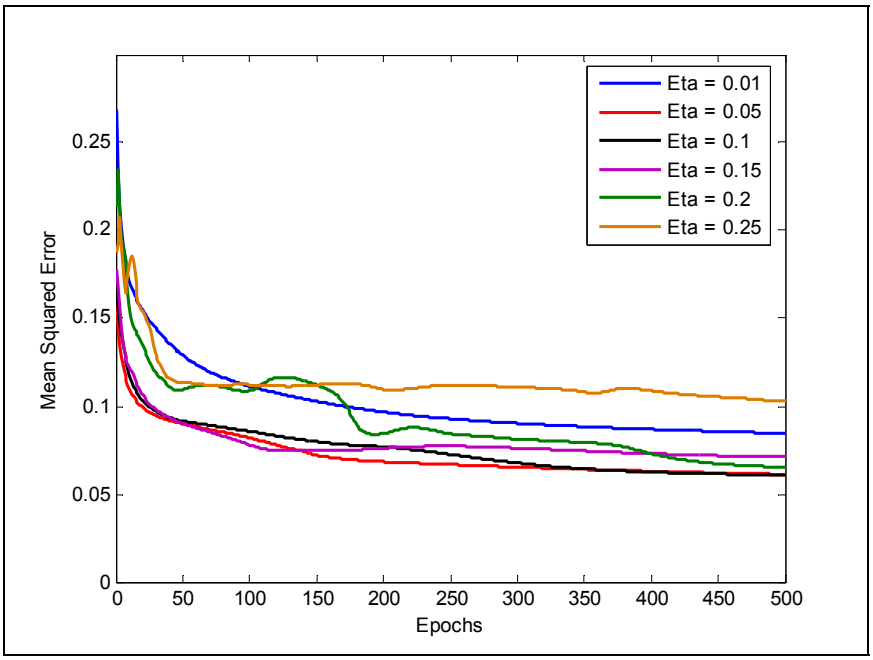


Figure 31: Results of different eta values

8. Solution Comparisons

The neural controller produced by the genetic algorithm approach contained two separate recurrent neural networks organised into layers. The bottom layer controlled the collision avoiding behaviour and consisted of 6 input sensors, 4 hidden nodes and 2 output nodes. The next layer controled the flag retrieval behaviour and consisted of 5 inputs (4 sensors, plus boolean input), 4 hidden nodes and 2 outputs. The controller produced by the backpropagation approach consisted of 9 inputs (8 sensors, plus boolean input), 14 hidden nodes, and 2 outputs. Figures 32 and 33 show these controllers.

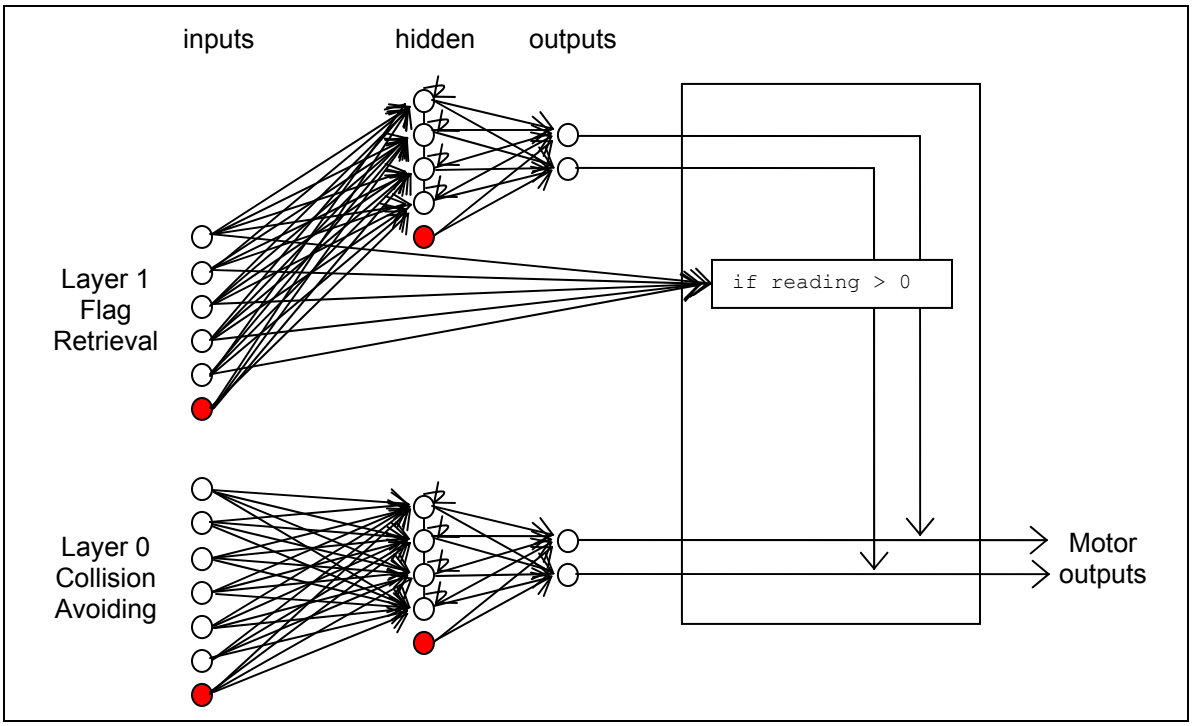


Figure 32: Control system produced by genetic algorithm combined with hard coded subsumption process, red nodes are bias nodes.

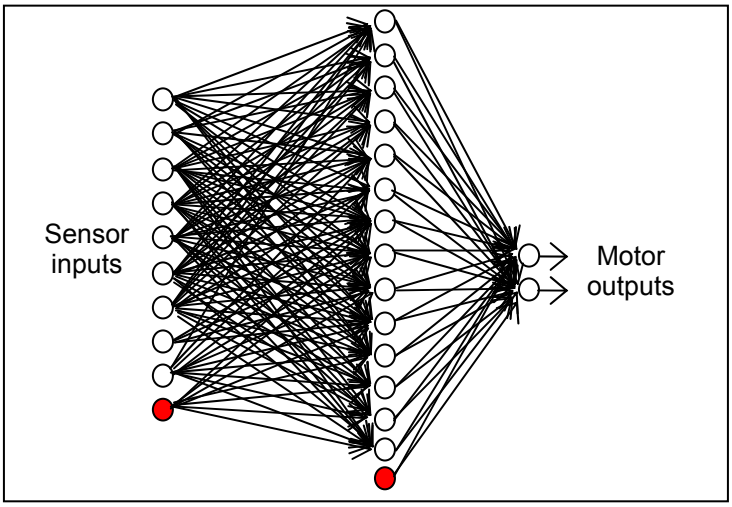


Figure 33: Control system produced using backpropagation

8.1 Tournaments

There are 3 tournament maps in the program, each one is rotationally symmetrical and both teams each have 5 members which start in equivalent positions. This makes the tournament fair for both teams. The 3 maps are shown in Figure 34.

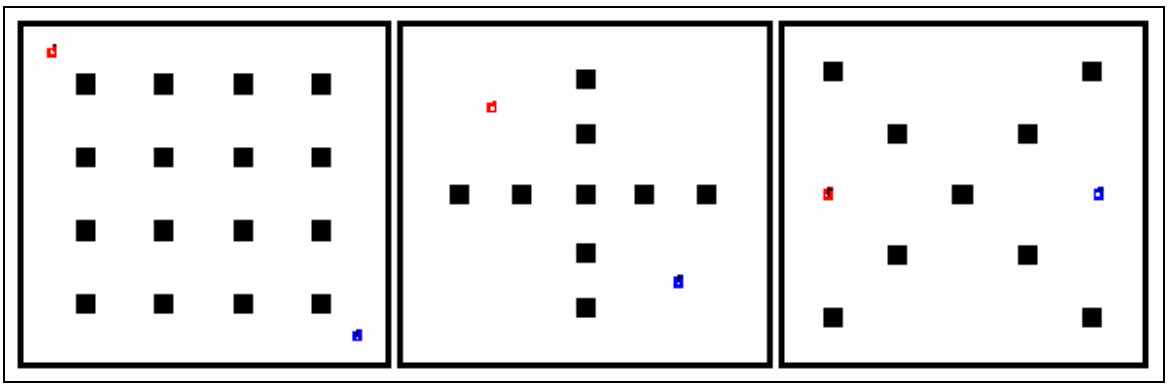


Figure 34: The rotationally symmetrical tournament maps

These maps will be referred to as the "grid" map, "+" map, and "x" map respectively for the rest of this section. The two techniques competed against each other in each of the maps for 5000 time-steps and were compared to see which performed the best overall. Each team had 5 agents all controlled by copies of the controllers produced, so each member of a team performed in the same way. The red team used the GA approach, and the blue used the backpropagation approach.

When tested in the "grid" map the resulting score was 4 – 2 for the red (GA) team. Figure 35 shows the programs graph data about how many collisions each team member had and how many times they captured the flag. As the results show, on average the blue (backpropagation) team had more collisions than the red team, however this was due to robots 2 and 5 having a large number of collisions over the 5000 time-steps, with robots 3 and 4 not having a single collision. Every member of the red team on the other hand did have at least 2 collisions. Robot 5 on the blue team was the only member that captured the opponent's flag, and succeeded twice. When this data is compared with the trail map (Figure 36), which shows the trails of every robot over the 5000 time-steps, it becomes apparent as to why these results occurred.

The robots 1, 3 and 4 from the blue team had a low number of collisions as they got themselves stuck running around in circles. The robots 2 and 5 are both stuck up against the right hand wall of the map. The red team spent the majority of the time running around the edge of the map, keeping a set distance from the walls. This means that they ran into both flags regardless of whether they intended to, so potentially captured the flag unintentionally.

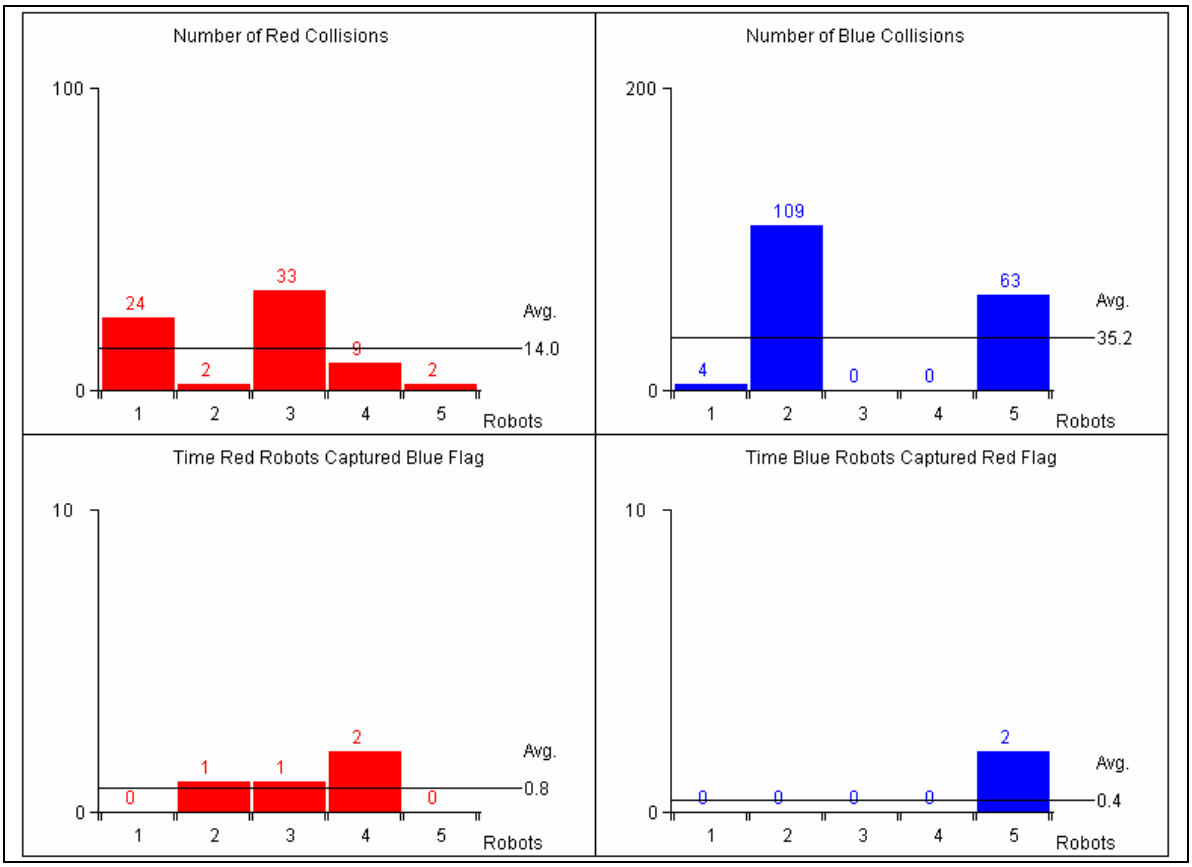


Figure 35: Data from the "grid" map, red is the GA team, blue is the backpropagation team

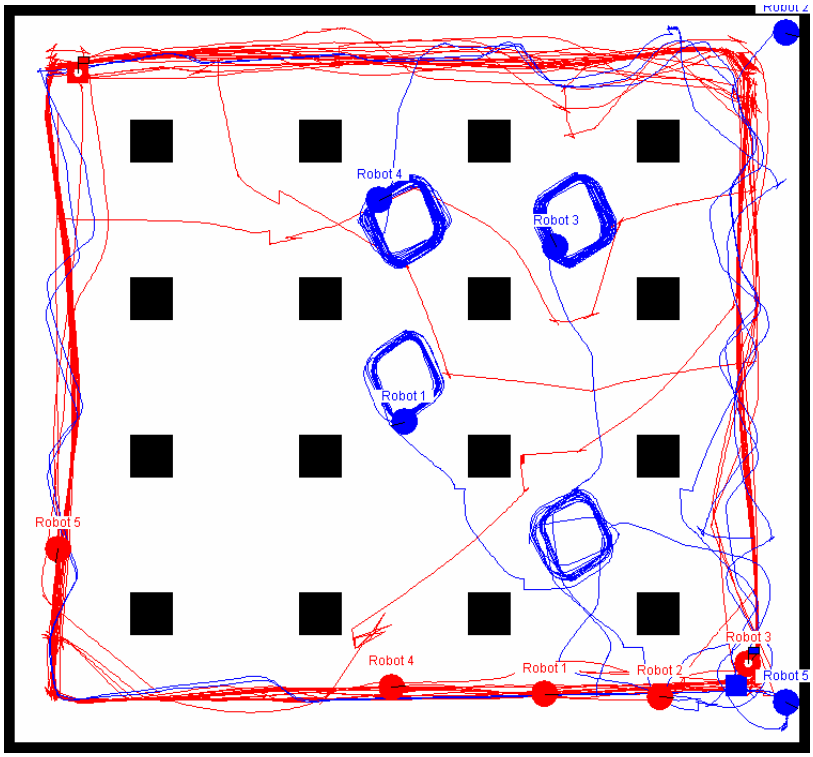


Figure 36: Trails from the "grid" map

When tested in the "+" map the resulting score was 7 -1 to the red (GA) team, which is a significant result for the genetic algorithm approach. Figure 37 and Figure 38 shows the results. The blue team had a higher average number of collisions than the red team with an average more than 3 times higher. Robot 5 of the red team had the worst number of collisions with 41, which was less than the minimum number of collisions on the blue team. The trail map shows that the red team performed in a similar way to the previous test in that they tended to follow the outside walls of the map, except for when they moved inwards towards the flag. It also shows that robot 3 on the blue team was close to scoring a second point for the blue team. Although not obvious in Figure 38, there are zigzag blue trails in the top left and right corners of the map, which show where a blue robot spent time bumping into the wall and gradually moving along it.

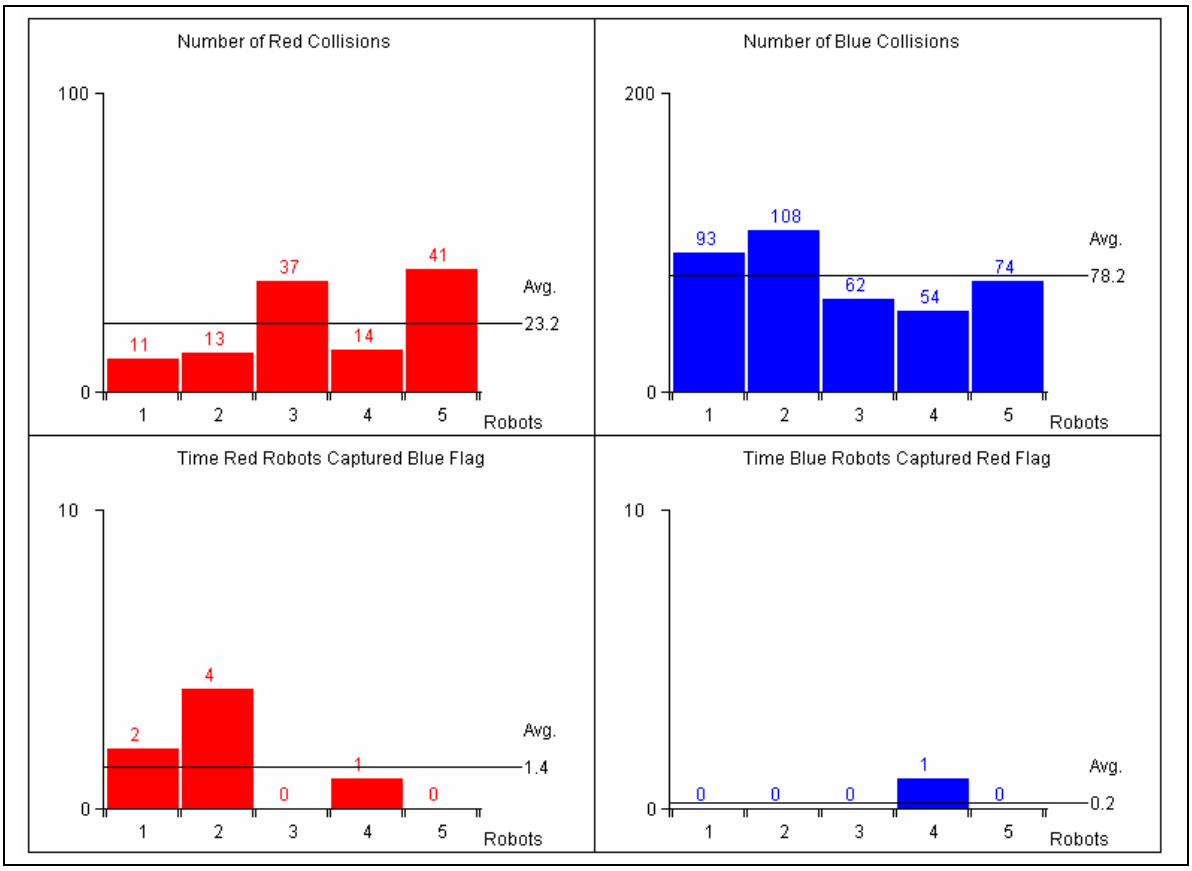


Figure 37: Data from the "+" map

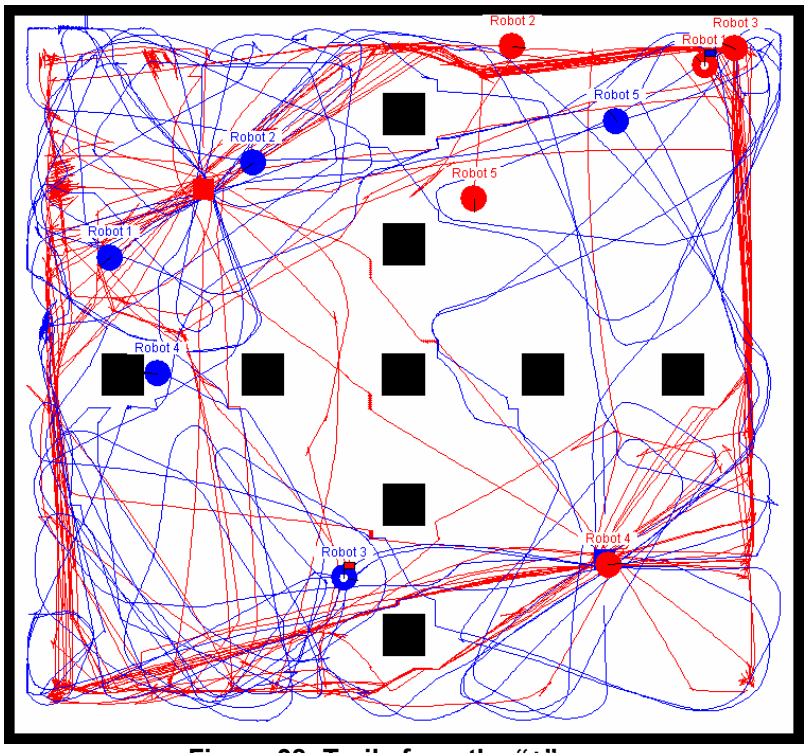


Figure 38: Trails from the "+" map

The result from the "x" map was 5 – 2 to the red (GA) team. Figure 39 and 40 show the results. This time around the average number of collisions for both teams was similar, however what can clearly be seen is that every robot on the blue team had a similar number of collisions, whereas the red team's robots varied quite a bit. Robots 2 and 4 on the red team had only 4 collisions each, yet the other 3 had significantly more, with robot 5 having a total of 51 collisions. In the top right corner of the map there is a cluster of red trails showing that at least 2 robots had an altercation, where they did in fact hit each other several times as they tried to get around each other. Again what is obvious from the trail map is that the red team tended to stick close to the outside walls except for when they saw the flag and moved towards it.

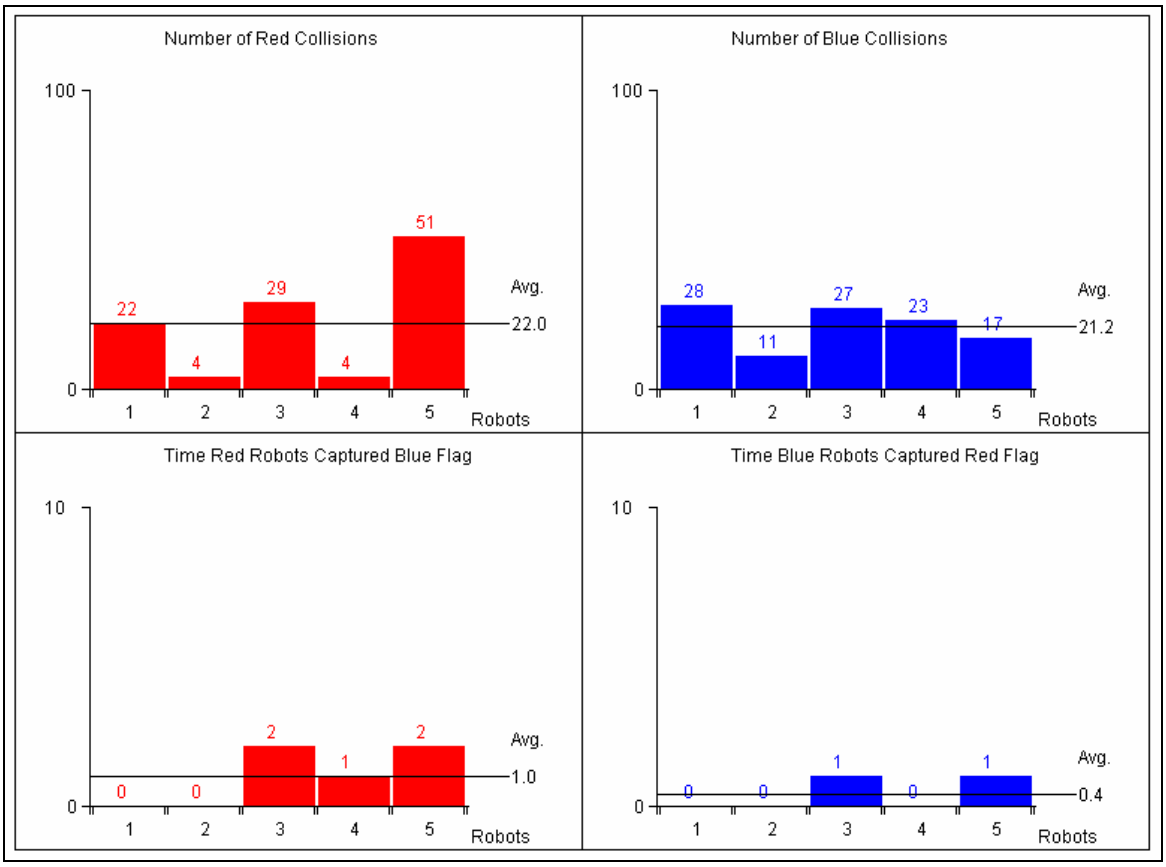


Figure 39: Data from the "x" map

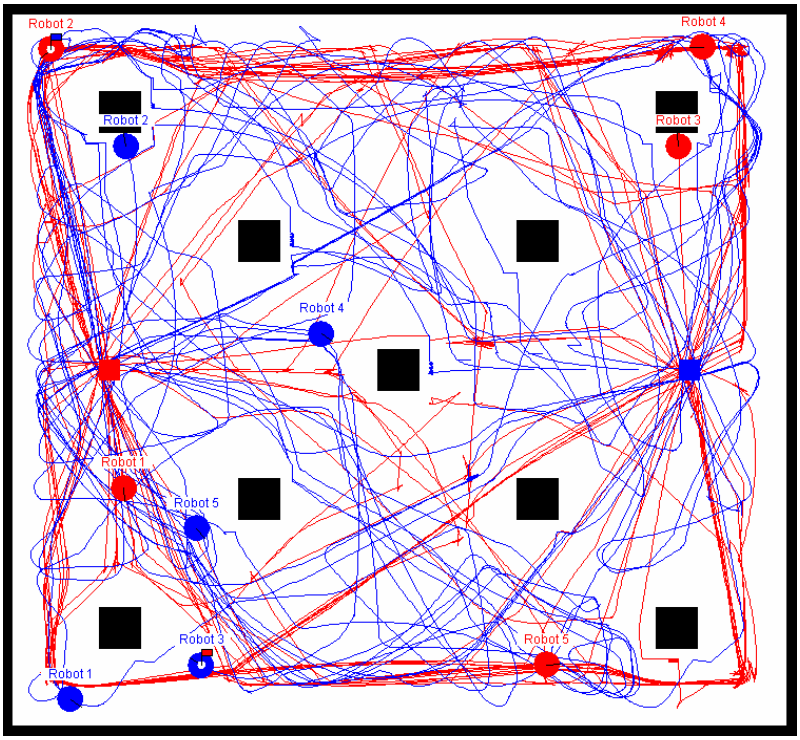


Figure 40: Trails from the "x" map

8.2 Results

The genetic algorithm approach won all of the tournaments by a good margin in each case. It is clear from the results that each technique approaches the task in different ways and the solutions they produce vary quite a lot.

The genetic algorithm approach has a generally much better method of collision avoiding, getting stuck less often than the backpropagation approach. The genetic algorithm approach to collision avoiding is to find the outside wall and stick a set distance away from it following it around. This way it never hits the outside walls and only has problems when it comes to other agents or the smaller obstacles. It only diverts from the wall when the flag retrieval layer takes over when the flag sensors are triggered. The trail maps also show that the GA approach likes to move in straight lines and makes quite sharp turns to put an obstacle to one of its sides so that it doesn't hit it.

The backpropagation approach is very different. This approach seems to have more trouble when dealing with obstacles. The trail maps show that generally when encountering a wall slightly off to one side the robot always makes a 90 degree turn in the opposite direction. The best example of this is in Figure 36 where 3 of the blue robots got stuck in loops in the middle of the map. This is the cause of the low scores for this team too as whenever the sensors detect a wall to the side the robot makes a 90 degree turn, even if it is not actually going to hit it. This means that when the robot is moving towards the flag, if it then detects a wall off to the side it will turn in an attempt to avoid the wall but as a result will lose sight of the flag.

The genetic algorithm approach is also able to get itself unstuck when it hits a wall as the recurrency in the layers start to work. It appears that if the network keeps receiving the same inputs this causes the network to change its outputs briefly causing a different movement to occur, which can lead to the robot getting unstuck. The backpropagation approach does not use any recurrency and so is not able to get itself unstuck, so once it is stuck it could potentially be stuck that way forever as it will consistently produce the same outputs for the same set of inputs.

The main problem with the outcome of the backpropagation approach was the difficulty in programming an agent to produce the training data. Programming a controller based on sensory inputs is quite complicated considering the amount of variation in the inputs. It was hoped that the backpropagation learning technique would produce a network able to generalise well enough to produce a reasonably good collision avoiding behaviour. However it turned out that the resulting controller was only ever going to be as good as the programmed agent, and unlikely to be any better. The GA approach, however produced much better overall general behaviour, and was especially good at collision avoiding. It did tend to adopt a more wall-following approach to collision avoiding; in that if it made sure it was running parallel to an obstacle then it was never going to hit it. The collision avoiding behaviour that was evolved turned out to be much better than the behaviour learned by the network trained from a programmed agent's data. This shows that genetic algorithms are useful in solving tasks such as collision avoiding, which aren't as easy to solve as initially thought, and produce solutions that may not have even been thought of if the controller were to be designed by hand.

Complexity of the resulting control systems is also important. The backpropagation approach used a single network with a large number of connections due to requiring a

large number of hidden nodes. Both behaviours of collision avoiding and flag retrieval were produced by this single network, whereas these behaviours needed to be separated into two networks in order for the genetic algorithm to produce a good solution. As a result of this though the number of network connections was greatly reduced, where the network produced using the backpropagation approach had a total of 170 weighted connections, but the controller produced using the genetic algorithm approach had only 72 connections across both networks. This makes the controller less complicated and slightly faster when calculating the next set of outputs.

The final approach used with the genetic algorithm did break away from the original idea in order to achieve better results as the network seemed reluctant to learn, due to the overly noisy environment and a simple fitness function. If these problems were solved then the results imply that a controller evolved with a genetic algorithm would produce a good solution to the task of capture the flag. The backpropagation approach could potentially also produce good results, but require training from an already working agent, which seems somewhat wasteful. If an agent has already been produced for the task, it is not really necessary to train a new one based on it.

The biologically inspired genetic algorithm seems a more appropriate learning technique to use for producing controllers capable of playing capture the flag, firstly in that it produces better results in general, but also because of the nature of the task being similar to what animals have to be able to do, such as being able to avoid collisions, and find food (or flags).

9. Limitations and Extensions

There are many limitations to the approach used throughout this project. Firstly the engineering type method that had to be introduced in order to efficiently deal with the subsumption process in the GA controller did not really stick to simply using a genetic algorithm to solve the task. The resulting controller therefore was not produced entirely by a genetic algorithm, and was also fairly predictable in that the instant the agent saw the flag; the controller switched its layer of control, even if it was likely to collide with a wall in the next couple of time-steps.

There is, however, an advantage to this approach, in that if layers are evolved to do different tasks, then this gives the potential to produce agents with different behaviours by layering the desired behaviours together using this approach. For example, in the future, if a new agent is needed for a completely different task such as producing a map of its environment, then the collision avoiding layer can still be utilised, and these previously evolved layers can be treated as plug-ins to the controller, meaning that only the new behaviour need be evolved.

Other limitations include the lack of recurrency in the backpropagation approach and the need for a pre-programmed agent to provide the process with training data. If the network had recurrency then a different backpropagation approach would need to be used as in [1], but this requires more memory in order to work, however the resulting controllers may be better able to get themselves unstuck in a similar way to the way the GA controllers could. Backpropagation techniques also exist that don't require prior knowledge of the exact outputs required, rather just an idea of whether the network produced the right output or not [6]. If this technique had been used then the requirement of a pre-programmed agent is removed.

The method doesn't use team-work at all. In fact each agent has no idea of the existence of any other agents in the environment, as it simply treats them as obstacles. A competitive co-evolution method, as used in [7, 8], may have been better as team approaches to solutions may have been developed. This would require separate controllers for each member of each team in order to produce the best overall output. If the agents were given an idea of, if not their own team, at least the opposition then attempts could have been made to allow for the agents to attempt to block there flag from being taken by moving in front of an approaching opponent.

There are many extensions that could be implemented, and the problem was lack of time, rather than lack of potential methods and approaches. Apart from the extensions mentioned previously in order to improve the limitations, there are many other extensions. For example, all obstacles in the environment are the same shape and size; would varying these produce different results, and different behaviours to become apparent? Comparisons between these approaches and other techniques could have been performed, as well as comparisons between two different methods of the same technique, with each method using different parameters. Only one set of sensor layouts was used for each approach, and would varying these have improved behaviour? With a random set of sensor layouts where the sensors aren't symmetrical on each side of the agent, could the approach still produce an agent that performs well enough?

There are also many extensions to the actual task at hand. For example the agent could not only try and find the flag but also build a map so that once it has been found the agent knows where it is, and could leave a trail that other agents could follow, similar to ants with their pheromone trails. If the environment was more realistic in terms of physics then the resulting controllers could be downloaded onto physical robots and tested in the real world to see how well they perform. The program itself could also be extended in order to include a wider number of techniques and tasks that could be explored and compared.

There are potentially hundreds of extensions that could be explored, but the approaches should always attempt to build on what has already been done, and possibly try other comparisons between genetic algorithms and backpropagation.

10. Conclusions

The aim of the project was to compare the two techniques, backpropagation and genetic algorithms when it comes to the task of producing an agent capable of playing capture the flag. The system built for this task, although relatively simple, worked really well and allowed the training and comparison of these two methods. The results showed that the overall best controller was that produced by the genetic algorithm, which implies that this approach is better suited to this task. That is not to say that the backpropagation technique is a bad approach, it is just better suited to other tasks, and is possibly not best suited for producing networks running in real-time.

The project shows that there is a large advantage in using genetic algorithms over techniques such as backpropagation if the task to be solved is known, but how to go about solving it is less obvious. The backpropagation approach used in the project required a knowledge of how the task should be achieved by passing in input-output data, whereas the genetic algorithm approach used the fitness functions to determine how well an agent was doing, without specifying how it should be doing it, which gives more flexibility and allows solutions to be produced that may not have previously been considered.

Although the genetic algorithm is only loosely based on its biological counterpart, it can be seen that evolution is a good way of producing a solution to a task and evolving behaviours such as the ability not to crash into anything or find a flag/ food/ water. The backpropagation approach struggled to generalise to new data very well and this is probably due to the complexity of the task itself, with potentially millions of different sensory input combinations, and a training set needs to be huge in order to allow the network to produce good generalisation ability.

The project demonstrated that while only attempting to produce relatively simple behaviours, this in itself is not as straightforward as first imagined, as many different variables and other information (such as environment complexity), can easily cause a negative effect on the approach being used. Producing more complex behaviours therefore requires an understanding of these simpler tasks first before trying to advance. The project also demonstrated that when the behaviours were produced, they used really quite simple networks, showing that these behaviours can be produced with only simple networks/ “brains”. This is a blessing, as simpler networks are easier to analyse and are also much more efficient, especially when there are many agents contained within the simulation all trying to calculate their outputs at once. This implies that the way in which simple animals perform tasks such as collision avoiding may not be as complicated as first imagined, it may just be a reaction to something close by rather than a complicated calculation of how far away the object is and how fast they are moving.

The project also showed that the subsumption approach had to be modified in order to get good results out of the system. It showed that getting the balance right between environment noise and fitness function complexity is somewhat of a black-art, and that if implemented incorrectly can produce very poor results where the agents do not learn at all.

11. References and Bibliography

[1] Boden, M. (2001). A Guide to Recurrent Neural Networks and Backpropagation. School of Information Science, Computer and Electrical Engineering, Halmstad University. [online] available from: http://citeseer.ist.psu.edu/507882.html

[2] Brooks, R. (1985). A Robust Layered Control System for a Mobile Robot. *IEEE Journal of Robotics and Automation, Vol. 2, No. 1, March 1986, pp. 14–23*. Artificial Intelligence Laboratory, Massachusetts Institute of Technology.

[3] Cummins, F. (1999). Recurrent Networks. [online] available from: http://cspeech.ucd.ie/~fred/teaching/oldcourses/ann98/rnn1.html

[4] Gomez, F., Miikkulainen, R. (1996). Incremental Evolution of Complex General Behavior. *Technical Report AI96-248*, Department of Computer Sciences, University of Texas, Austin. [online] available from: http://citeseer.ist.psu.edu/ gomez96incremental.html

[5] Harvey, I. (1996). The Microbial Genetic Algorithm. School of Cognitive and Computing Sciences, Univeristy of Sussex, Brighton. [online] available from: www.cogs.susx.ac.uk/users/inmanh/Microbial.pdf

[6] Henaff, P., Delaplace, S. (1996). Using Backpropagation Algorithm for Neural Adaptive Control: Experimental Validation on an Industrial Mobile Robot. *Theory and Practice of Robots and Manipulators (ROMANSY 11)*, Udine, Italy, July 1996.

[7] Nelson, A. L., Grant, E., Barlow, G., White, M. (2003). Evolution of Complex Autonomous Robot Behaviors using Competitive Fitness. Center of Robotics and Intelligent Machines, Department of Electrical and Computer Engineering, North Carolina State University. [online] available from: http://citeseer.ist.psu.edu/667690.html

[8] Nelson, A. L., Grant, E., Henderson, T. C. (2003) Evolution of Neural Controllers for Competitive Game Playing with Teams of Mobile Robots. *Robotics and Autonomous Systems 46 (2004) pages 135-150.*

[9] Russel, S., Norvig, P. (2003). Artificial Intelligence A Modern Approach, Second Edition. New Jersey, Pearson Education Inc.

[10] Sun MicroSystems, Inc. (2004). Java 2 Platform Standard Edition 5.0 API Specification. [online] available from: http://java.sun.com/j2se/1.5.0/docs/api/

[11] Togelius, J. (2003). Evolution of the Layers in a Subsumption Architecture Robot Controller. Evolutionary and Adaptive Systems, Department of Informatics, University of Sussex, Brighton. [online] available from: http://julian.togelius.com/ Togelius2003Dissertation.pdf

[12] Togelius, J. (2004). Evolution of a Subsumption Architecture Neural Controller. *Journal of Intelligent & Fuzzy Systems 15:1, pages 15-20.* [online] available from: http://julian.togelius.com/Togelius2004Evolution.pdf

[13] Westbrook, D. L., Atkin, M. S., Cohen, P. R. (1996). Capture the Flag: Military Simulation Meets Computer Games. Experimental Knowledge Systems Laboratory, Department of Computer Science, University of Massachusetts, Amherst.

Appendix A – Professional Considerations (identified before starting the project)

This section lists the guidelines and regulations set out by the British Computer Society (BCS) in their Code of Conduct and Code of Good Practice that are relevant to this project. Due to the research nature of the topic few of the Code of Conduct guidelines apply.

The paragraphs from the Code of Conduct affecting this project are:

- *"You shall seek to upgrade your professional knowledge and skill, and shall maintain awareness of technological developments, procedures and standards which are relevant to your field."*

Throughout this project I will read many papers relating to the topics of artificial life, neural networks, and genetic algorithms to help aid my understanding of the area. I will also try and keep up to date with any recent advances or research in the field.

- *"You shall not claim any level of competence that you do not possess. You shall only offer to do work or provide a service that is within your professional competence."*

Although I haven't previously completed a task involving evolving of artificial agents, I have completed assignments relating to the areas involved in this project. I have produced programs that use genetic algorithms, neural networks and back-propagation. I have also used Java for 2 years, and have had experience at creating GUIs using Java. I therefore believe that I have the ability to complete this project by combining all of these skills.

- *"You shall accept professional responsibility for your work."*

All of the work produced during this project will be of my own doing unless stated otherwise, and anything that isn't mine will be properly referenced.

- *"You shall observe the relevant BCS Codes of Practice and all other standards which, in your judgement, are relevant."*

I will follow any relevant guidelines in the Code of Practice including:

- Producing well structured code and following any programming guidelines and structures specific to the Java language.

- Produce code that is easy to understand.

- Honestly summarising mistakes, good fortune and lessons learned.

- Recommending changes that will benefit extensions to the project.

Appendix B – Project Log

Week 1, Autumn Term, 2006

Spent this week thinking about what area I wanted to work in and decided on focusing in on artificial evolution, neural networks and A-Life. Spoke with several potential supervisors and my academic adviser about what exactly to focus in on and what advice they could give.

Week 2, Autumn Term, 2006

Decided on developing a simple capture the flag simulation to compare a genetic algorithm with a training method such as back-propagation and read several papers related to the topic. Looked at java and Matlab to determine which would be the best language to create the simulation in, and decided on java as I have more experience of this language. Registered with supervisor and met up to discuss my proposal.

Week 3, Autumn Term, 2006

Wrote up the aims and objectives of the project and how it relates to my degree programme. Handed project proposal to supervisor and discussed the project further. It was suggested that I take an existing simulation and use it to do the research.

Spent the rest of the week playing with java's graphics packages to see how easy it was to create simple moving shapes and how the double buffering technique works to stop animation flicker. After figuring this out it was decided that I would build the simulation environment myself.

Week 4, Autumn Term, 2006

Spent the week creating a simple two-dimensional Braitenberg vehicle simulation in java. This involved figuring out how to translate left and right wheel speeds into turning and forward motion on a 2-d coordinate system of a computer monitor and how to implement simple sensors. Finished off with a simple vehicle that would chase another one around the screen.

Week 5, Autumn Term, 2006

Met with supervisor to discuss what exactly could be put in the analysis phase of the interim report for a research based project. Spent the week reading more papers and optimising the Braitenberg vehicle program so that the classes can be used in the actual project without having to change them much.

Week 6, Autumn Term, 2006

Started to think about what classes would be necessary for the simulation, however due to other coursework deadlines didn't do much this week.

Week 7, Autumn Term, 2006

Wrote the introduction to the field, as well as the aims and objectives of the project. Also did a bit of research into what kinds of neural networks are suitable for the control systems of each robot. Decided this would be determined once the simulation was working as I can then build several network architectures and plug them in as necessary.

Week 8, Autumn Term, 2006

Read the BCS Code of Conduct and Code of Good Practise and decided which rules and regulations apply to the project and wrote how I plan to stick to them. Focussed in more on the basic classes that would be necessary and what main fields and methods each class would require as well as how they collaborated. Wrote up all of this as the analysis phase of the interim report.

Week 9, Autumn Term, 2006

Properly wrote up the project plan into a neater format using a PERT chart ready for the interim report. Met with supervisor for half an hour to discuss the interim report to see if any improvements were necessary as well as to discuss progress so far and the object representation based technique I am intending to use for the simulation. The intention is to write the program over Christmas and have it working ready to do the research over the spring term.

Week 10, Autumn Term, 2006

Finished the interim report and handed it in. Started on the design, which involved breaking down what was discovered in the analysis phase into more depth, so that coding can begin as soon as possible over Christmas.

Week 1, Spring Term, 2007

Spoke to supervisor to inform them of progress over christmas break, and to continue building on the basic program developed over the christmas period.

Week 2 – 5, Spring Term, 2007

Lots of time spent trying to resolve the interference issue within the subsumption architecture when using the genetic algorithm approach. Numerous meetings with supervisor to discuss the cause and potential solutions. Eventually produced a successful controller after making a change to how the subsumption architecture worked.

Week 6, Spring Term, 2007

Developed the backpropagation side of the program and got a working solution. Noticed that the hand-coded agent used in the backpropagation approach wasn't very good so improvements were made to rectify this. Started to build the tournament side of the program complete with data capturing and scoring information.

Week 7 - 8, Spring Term, 2007

Finished a working version of the program and performed the research necessary to write the report. Saw supervisor to discuss requirements of the report and what should be included. Wrote up the introduction and background sections of the report.

Week 9 – 10, Spring Term, 2007

Wrote up the rest of the draft report, excluding abstract, professional considerations and appendices. Submitted draft report to supervisor for review.

Easter Holiday, 2007

Went through program code and tidied it up, including the removal of redundant sections of code that were no longer being used, and addition of comments to the code where necessary. Received feedback from supervisor on draft report and started to make changes ready for the final report.

Week 1, Summer Term, 2007

Finished improving report and prepared it for formal submission in Week 2. Started preparing presentation ready for Weeks 3 – 4.

Appendix C – CRC Cards (produced as part of the Requirements Analysis)

Sensor	
Responsibilities	**Collaborators**
Detect objects (Robots, Flags etc.) within its range and return a value.	Robot Flag Base

ANN	
Responsibilities	**Collaborators**
On receiving inputs calculate outputs. Receives weight adjustments from BackPropagation.	Robot BackPropagation

Robot	
Responsibilities	**Collaborators**
Requests sensor input. Update its position coordinates. Draws itself on the screen. Stores whether it has flag or not.	Sensor ANN Arena Flag

Flag	
Responsibilities	**Collaborators**
Draws itself on the screen. Informs Sensor of its position.	Arena Sensor

Base	
Responsibilities	**Collaborators**
Draws itself on the screen. Informs Sensor of its position.	Arena Sensor

Arena	
Responsibilities	**Collaborators**
Calls all objects methods to update their position and draw themselves.	Robot Flag

UserInterface	
Responsibilities	**Collaborators**
Contains the Arena object. Allows user to change options for GA and BackPropagation. Tells the Arena object to run the program.	Arena GA BackPropagation

GA	
Responsibilities	**Collaborators**
Contains arrays representing weight values. Creates new ANN objects using weight values. Receives input from the UserInterface class to set its internal properties.	ANN UserInterface

BackPropagation	
Responsibilities	**Collaborators**
Calculates errors for the ANN class and sends weight adjustments. Receives input from the UserInterface class to set its internal properties.	ANN UserInterface

Appendix D – System Overview and Screenshots

The final system is split up into 22 classes which are as follows:

UserInterface

This class deals with creating the actual user interface and contains all of the inner class listener required for controlling the system. The user interface has different modes which when selected display different controls and options on the screen (See Screenshot 1, 2 and 3 after class descriptions).

Arena

This class extends a Canvas object and deals with the drawing of the simulation on the screen, including holding all objects contained within the current simulation.

ArenaObject

This is an abstract class for objects that can be placed into the arena. It contains basic methods for setting and getting the height, width and coordinates of an object and contains a numbre of abstract methods which must be implemented for an object to be placed in the arena. Any object to be placed in the arena must extend this class.

CollidableObject

This is an abstract class which extends the ArenaObject class and contains a single abstract method for detecting collisions between this object and another. This class should be extended by any objects to be placed in the arena that require collisions to be detected.

Robot

This class extends the CollidableObject class and creates a robot which is visible on screen in the Arena. It contains a number of Sensor objects and a RobotController object which deals with the calculation of outputs. It contains a number of variables for keeping track of what the Robot has done, including the number of collisions it has had, the number of flags it has collected, the number of times it has reversed etc. It also contains methods for calculating its new speed and angle based on its current motor speeds, and methods for calling the RobotController to calculate its new motor speeds.

ArenaWall

This class extends the CollidableObject class and creates black rectangles or squares in the arena which act as walls. These cannot be passed through by Robot objects.

Flag

This class extends the ArenaObject class as collisions do not need to be detected, rather the Arena checks if a Robot is in proximity of a Flag object. This class deals with the

drawing of a base representation as a square with a flag icon if the flag is currently present at the base.

Sensor

This class creates a Sensor object of a certain size, angle and offset and deals with the detection of a specific type of class which is passed in at creation. It has methods for detecting if an instance of the detectable class is within range and returning a value between 0 and 1 depending on how close the object is. It also checks if there are any obstructions to the sensors view such as a wall in the way, causing the sensor to return a 0.

RobotController

This class has a Vector for holding a number of ANN objects and has methods for calculating the output of each ANN based on the sensor input values it receives. It then uses the subsumption process to determine which set of outputs from which ANN it should return. It also contains a method for cloning itself so that other RobotController objects can be produced with identical networks.

ANN

The neural network class which creates a 3 layer network with a user specified number of neurons. It can be recurrent or not depending on the situation it is being used in. It has methods for calculating the outputs given a set of inputs using the tanh function as the neuron activations. It also has methods for adjusting and replacing the weights in each of the weight layers.

GA

This class controls the genetic algorithm process and contains a population of ANN objects in an array. It has methods for evolving these networks based on their fitnesses using a roulette-wheel selection method with elitism. It also stores the average and best fitnesses of each generation in two arrays which can then be passed into a FitnessGraph object to plot the fitness change over time.

BackPropagation

This class performs the backpropagation learning algorithm on an ANN object. It gets passed a Vector of input-output mappings and has methods for randomly reorganising these and then splitting into two groups, training and generalisation sets. It has methods for calculating the error on the outputs given a certain set of inputs and can propagate this error backwards through the network to get the weight change values. It then calls the ANNs weight change methods to update the weights accordingly. It also stores the training and generalisation error produced by the graph at each epoch which can then be passed into an ErrorGraph object to plot the error change over time.

Fitness

This class deals with calculating the fitness of neural networks. It has several static methods for calculating the fitness for different situations including flag finding fitness,

collision avoiding fitness, and wandering fitness. Each method takes a Robot object as a parameter and retrieves the relevant information from this Robot object. It then calculates a fitness score and updates the ANN being evolved contained in the Robot object with its fitness score.

CodedRobot

This class extends the Robot class and overrides the direction method which normally calls the RobotController method for calculating the outputs and updates the direction of the agent accordingy. The new direction method gets the sensor readings and contains a series of 'if' statements to determine what the Robot should do next. This is the hand-coded agent used to gather the data required for the BackPropagation class. As a result this class contains a variable holding a DataCapture object.

DataCapture

This class contains a Vector data which is updated to contain new input-output data when the relevant method is called. It also has methods for saving the contents of this Vector to a text file and one for loading data into the Vector from a text file. Each of these methods has the relevant IOException error catching clauses.

LineGraph

Creates a window with a Canvas object and has methods for drawing graph axes, the legend and plotting data. It also has a method which saves the data to a text file. Although not an abstract class it should be extended so as to allow for different types of lines to be plotted.

FitnessGraph

This class extends the LineGraph class and takes two arrays. The first is the average fitness and the second is the total fitness across generations. It has methods for calculating what scale the lines should be plotted on based on how many generations there are, the size of the window and how much the values range by. See Screenshot 4 for a sample graph produced by this class.

ErrorGraph

This class extends the LineGraph class and takes two arrays as the FitnessGraph did. The first array is the training error and the second is the generalisation error across epochs. Similarly this works out the scale of the plots based on size and range of the data. See Screenshot 5 for a sample graph produced by this class.

LayerSetupUI

This creates a small window which allows the user to specify exactly how many sensors, of what type and size they want for their GA agents. It shows a view of an agent with the added sensors and allows the user to adjust the individual settings of the sensors with various sliders. It also allows the user to specify how many hidden nodes they want in the network for that particular layer. It also has a load button to allow the user to load in a previously saved layer. See Screenshot 6 for what this window looks like.

LoadSaveHandler

This class deals with the loading and saving of various data such as individual layers, entire evolved controllers, trained controllers. Its methods also catch exceptions and display the relevant error messages.

Quicksort

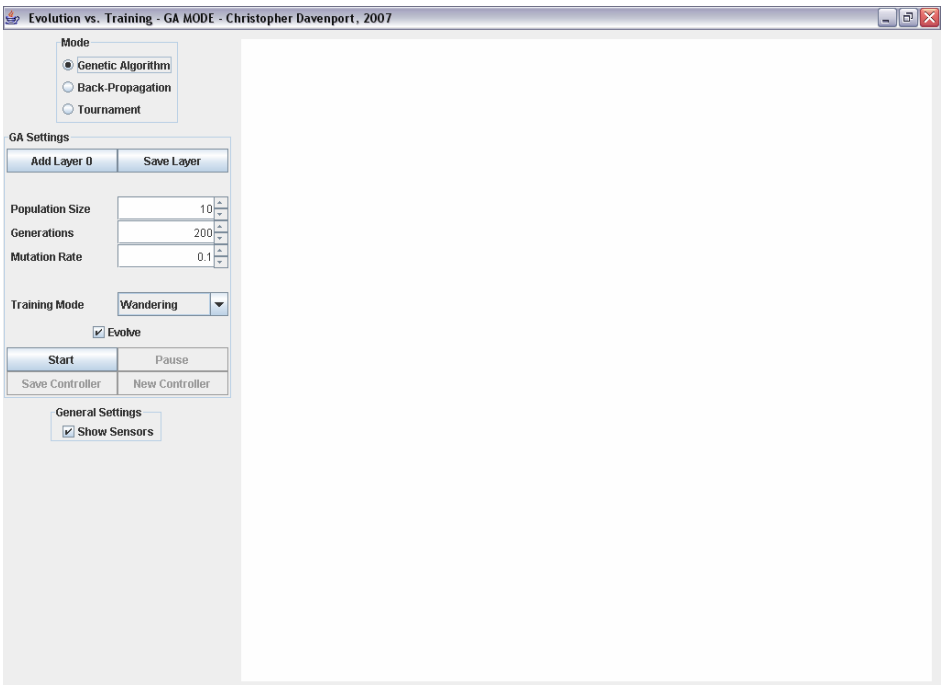
This class performs the quicksort algorithm on a Vector of data. It is called by the Sensor class to order objects according to distance from the Sensor object. It has a method sort which takes a Vector of data and returns the sorted data in another Vector.

TournamentStatistics

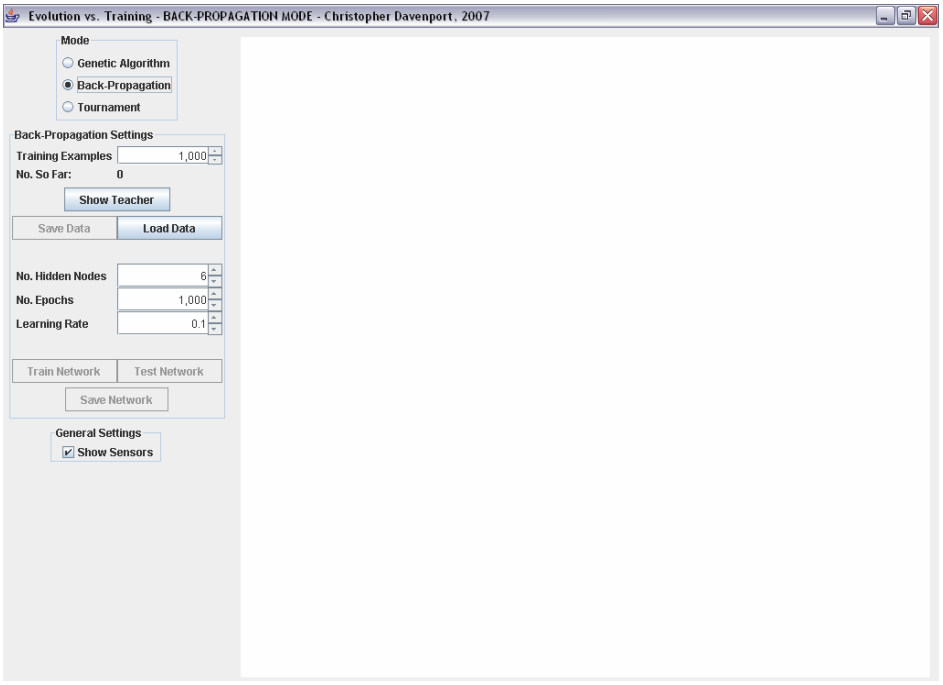
This creates a window containing 4 bar graphs representing the number of times each agent on each team had a collision or captured a flag. It has methods for drawing the axes of each bar graph, drawing the title and plotting the bars according to the data it has received. It has an inner class BarGraph which extends a Canvas containing an overridden paint method. See Screenshot 7 for what this window looks like.

Screenshots

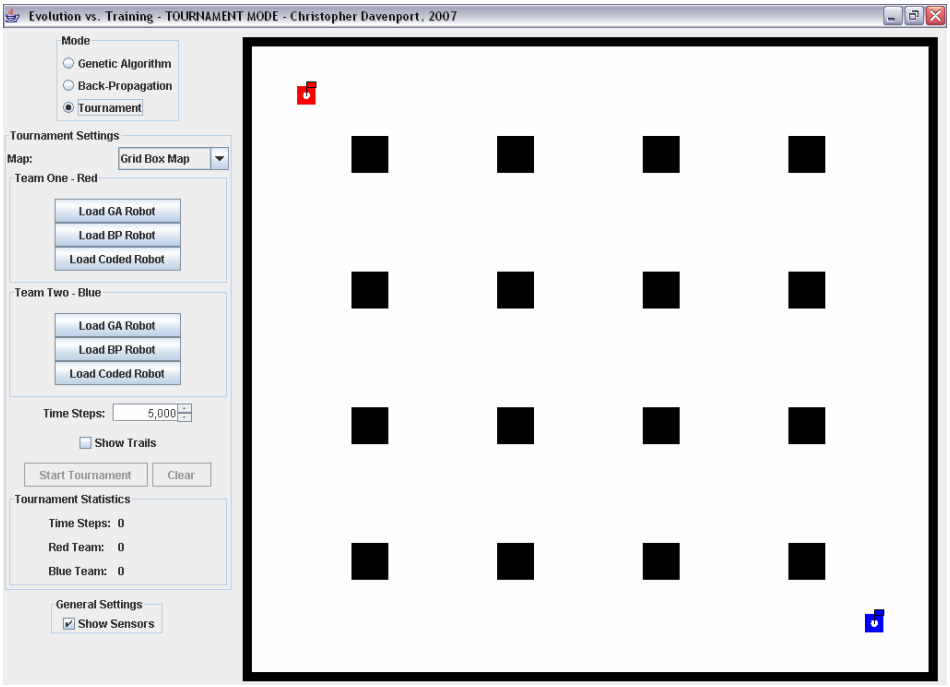
Screenshot 1 – User interface in GA mode



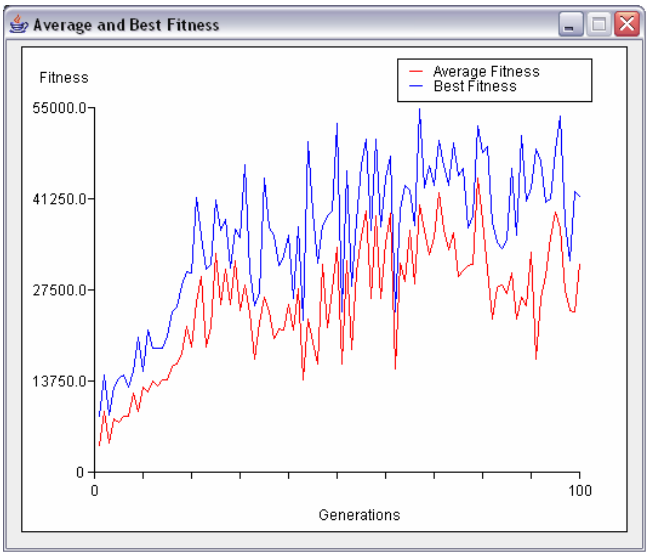
Screenshot 2 – User interface in Back-Propagation Mode



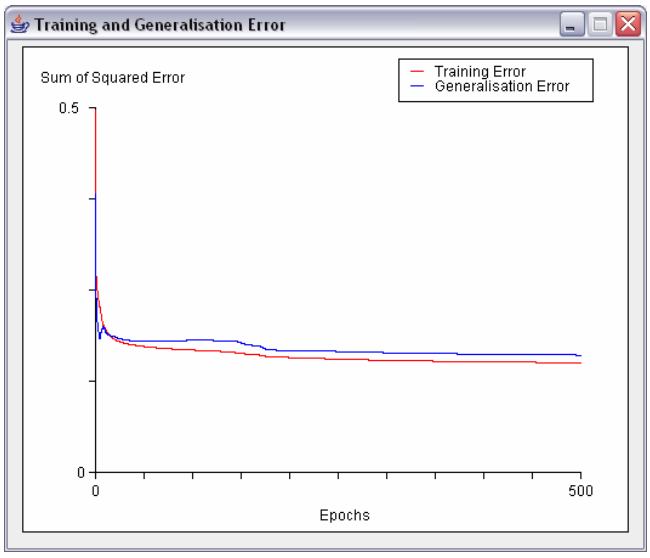
Screenshot 3 – User interface in Tournament Mode



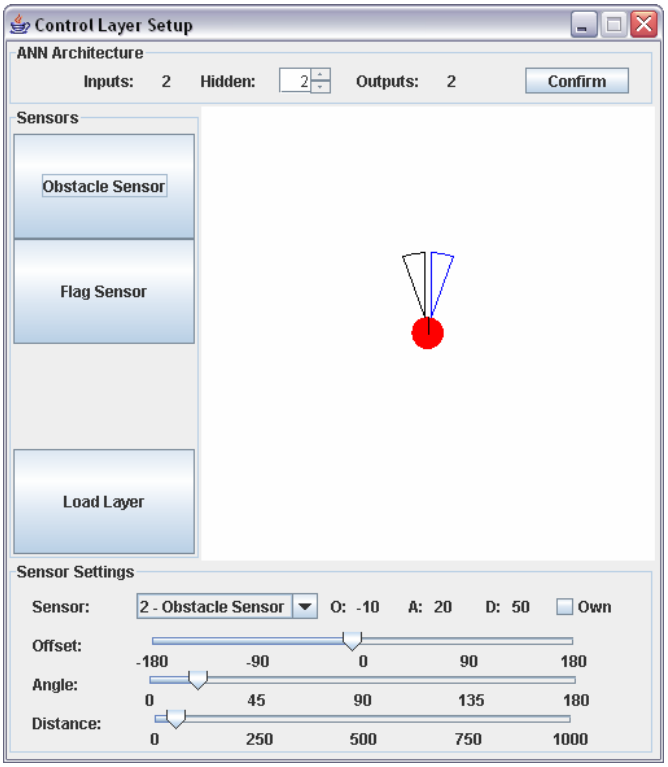
Screenshot 4 – Fitness Graph (noting that all fitness graphs in Section 6 were produced in Matlab, but this is how the data was displayed in the program itself)



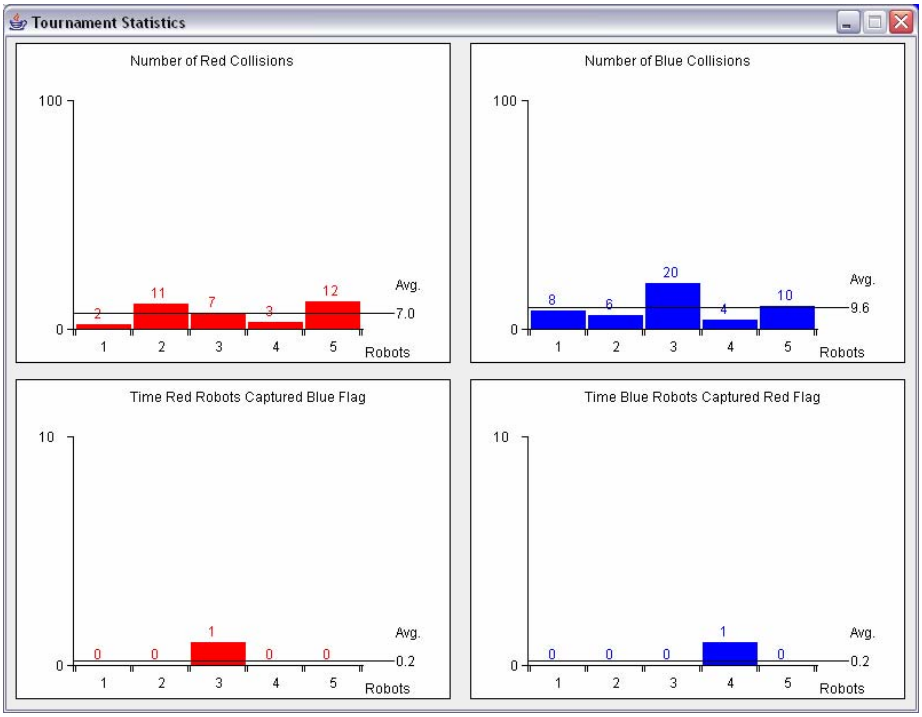
Screenshot 5 – Error Graph (noting that all error graphs in Section 7 were produced in Matlab, but this is how the data was displayed in the program itself)



Screenshot 6 – The layer setup window, allowing network architecture and sensors to be set up



Screenshot 7 – The tournament statistics window showing the results of the tournament



Appendix E – Source Code

The following pages contain the source code for the program. The order of the classes is the same as in Appendix D.

Class Contents