THE CO-EVOLUTION OF COOPERATIVE BEHAVIOUR

By Matthew Kelcey

Supervised by Dr. Peter Vamplew

A thesis submitted in partial fulfilment
of the requirements for a degree of
Bachelor of Computing with Honours

UNIVERSITY OF TASMANIA

1997

Abstract

Much work has been done on applying evolutionary techniques to a number of varying applications and in particular the training of neural networks. Most evolutionary systems though are aimed at solving tasks requiring only a single entity. This project applies co-evolutionary techniques to construct teams for multiple entity problems with a focus on the communication aspects required between team members.

Acknowledgments

I would like to thank my supervisor Dr. Peter Vamplew for his expert knowledge of neural networks and genetic techniques in general.

Thanks also to Id Software for honours stress relief entertainment software and Microsoft for the perfect lesson in how to write user friendly and intuitive software. :)

Table of Contents

Table of Figures

1 Introduction

1.1 Objectives

Evolutionary techniques in all their forms, such as genetic algorithms, genetic programming and evolution strategies, have been shown to give good results with a wide range of varying problems. In particular they have been able to evolve behaviours in simulated and real-time based controller systems, generally in the field of robotics. Most research studies have focussed on a single controller for a single entity performing the required task. Less work has been done on team based problems where a number of distinct entities are used to construct each possible solution. Any communication in these types of systems has been predefined and static.

The objective of this project is to incorporate the communication aspects of team based problems into the evolutionary system so that it can be evolved as an aspect of the behaviour of an individual. This has the advantage of allowing complex problem-specific communication systems to be evolved unique for each task. Once such a system has been developed it can be compared with human defined communication systems to decide whether evolving communication in this way can be useful.

All evolutionary systems also have the advantage that they can be coded somewhat independently of the problem being solved. As such it is an aim of this project to develop a system that requires the minimum knowledge of how the communication will act, only needing a definition of what behaviours to reward. In this way the desired behaviour of a problem can be abstracted away from the actual details of the underlying communication that will be involved.

1.2 Method

This project involves five distinct stages

1. The study of literature dealing with previous research in the areas of genetic algorithms, neural networks and the combination of the two.

2. The design and implementation of a simple generic evolutionary system incorporating a number of different implementation aspects and several new procedures[1].
3. The design and implementation of a simple feed forward neural network class for the simulation of possible controller solutions.
4. The refinement of the evolutionary framework to perform specifically on the neural network architectures defined in stage 3.
5. The testing of the system on increasing difficult tasks with conclusions on whether such an approach can give valid solutions within time and processing effort feasibility constraints.

1.3 Summary literature review

Evolutionary techniques are so called because of the conceptual similarities that exist between them and the general principles of natural selection and genetics. Such techniques work by maintaining a population of individuals, each of which is an encoded instance of a possible solution to the problem being solved. Techniques are defined for the recombination of these individuals that have been chosen by selection methods.

Co-evolutionary techniques are an extension to include maintaining a number of populations at once. One reason this is done is to provide solutions to problems that require a number of different parts, such as the co-evolutionary paradigm.

When applying such techniques to specific problems such as neural networks[2] a number of issues must be addressed. The resolution of these issues often involves the modification and specialisation of the evolutionary framework to work only on that type of problem.

[1] Including pairwise elitism, selective genetic operators, sub-population team based management and guessing-based selection

[2] And in particular neural networks as a means of defining some form of controller.

2 Relevant literature

2.1 Evolutionary Techniques

2.1.1 General Concepts.

Evolutionary algorithms use principles described in natural selection and genetics as the basis of an adaptive searching technique. Luger and Stubblefield (1993 :529) describe the genetic algorithm as an “implementation of a powerful form of hill climbing that maintains multiple solutions, eliminates unpromising solutions and improves good solutions.” Since they are a parallel search method they are proficient at quickly finding near optimal solutions for domains whose state space consist of many local minima. Though the execution of evolutionary techniques can be slow Wasserman (1993 :74) states “In the long run, this is probably not a valid objection. These algorithms, like neural networks, are parallel in nature; their execution rate increases almost linearly with the number of processors.” Schultz (1994 :3) also points out that “…because of the nature of the genetic algorithm, the initial knowledge does not have to be very good; it only needs to make the system have an occasional success at performing the task.”
If an approximate solution found by the system is not accurate enough quite often more traditional methods will converge on a final solution faster. In this way hybrid combinations of evolutionary techniques and other search methods may produce more efficient results.

Once a problem is clearly defined an encoding for each possible solution needs to be chosen for the evolutionary technique to work on. This is usually a vector termed the *chromosome*. A way of converting this chromosome (the individual’s *genotype*) into a potential solution (the individual’s *phenotype*) needs to be defined, unique for the encoding and the problem. A fitness function is also needed that assigns a real value to each solution based on its relative ability to complete the objective. The allocation of a fitness function is a non-trivial task[3] and its definition will have a great outcome on the performance of the overall algorithm. An evolutionary technique works on a

[3] For non-trivial problem domains.

population of initially random possible solutions[4] and at each time step defines a new *generation*.

The three genetic operators are selection, recombination and mutation. Using these methods individuals of the population are chosen proportional to their relative fitness and recombined to create a new population whose overall average fitness is greater than the last generation. There are a number of issues though which must be addressed for each unique problem. These include the following.

- *Representation:* evolutionary algorithms work with "genetic" representations of trial solutions, usually in form of a string of real or integer numbers. The user has to provide a suitable representation and a function that maps genetic representations into phenotypic trial solutions
- *Performance:* a function has to be provided that associates a performance value with each individual. The performance should reflect how good or how useful the individual is to solve the considered problem.
- *Creation of offspring:* the user has to specify operators (eg. crossover or mutation) that allow the creation of new individuals given one or two parent individuals. Very often these operators need repair functions to ensure that the offspring is a valid trial solution, or they include local hill climbing to speed up the local fine-tuning.

  (Branke, 1995 :2)

Functions such as crossover and inversion use information already in the population as a means of generating better solutions. Mutation techniques introduce new information about the search space into the system and ensure that the system both is able to reach every location in the search space and will not always become stuck in local minima. Since evolutionary algorithms have been shown to be poor local fine tuners (Yao, 1996) (Branke 1995) hybrid approach's using local gradient search based methods can in certain conditions outperform either used alone. Usually the method is

[4] Some pre-processing can be done on the initial solutions to aid the algorithm's performance.

to apply the genetic technique until there is some manner of convergence and then switch to a local hill climber (such as back propagation)

Procedures for implementing these methods in terms of a genetic algorithm were first introduced by John Holland (Holland, 1962) and A. Fraser (Fraser, 1962) working independently with few differences. The main difference between their early work was "...Holland suggested reproducing each parent in proportion to its relative fitness." (Fogel, 1996 :90)

2.1.2 Selection Methods

It is important to make fitness evaluation a function that is as continuous as possible so that the genetic operators can correctly discriminate between the different levels of fitness in the population, even so a non-continuous function can provide valid results. After fitness evaluation the raw fitness values must be converted to some scaled fitness values ready for the selection process. For example if minimising an objective function then small-raw functional values should be mapped onto high-scaled fitness values for selection.
A technique such as roulette selection requires that all fitness values are positive and adding any constant to remove negative values will scale the values unevenly, making the selection act as a random function.
A number of approaches to converting the fitness from raw values to scaled values exist and can be used alone or in combination.
Masters (1993) gives the example of the function to map fitness values from low-raw to high-scaled values, $F(v) = e^{Kv}$ for some negative constant K. He states for values of $v \ni [0..1]$ from experimentation using $K = -20$ is effective.
Goldberg (1989) scaled all fitness values relative to the mean fitness of the entire population to make the maximum fitness a predefined constant multiple, **k**, of the mean. By experimentation he claims that rescaling with **k** between 2 and 1.5 gave robust results.

Another problem associated with roulette selection is that the best chromosome in any population can be lost in a generation through chance.
One way to ensure asymptotic convergence towards a global maximum is to apply a heuristic such as *elitism selection* (Grefenstette, 1986) where the fittest individual in

each generation is copied to the next generation unchanged. The actual rate of convergence though varies for each application.

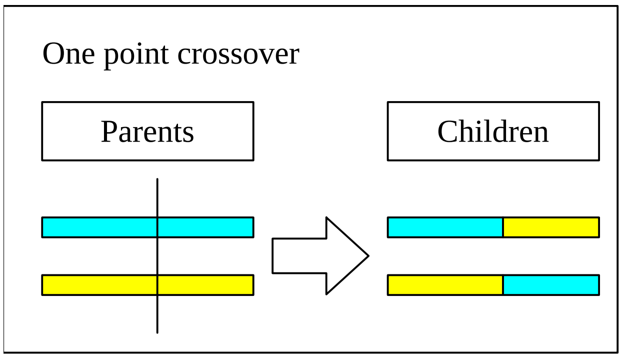
Masters (1993) used a technique based on roulette selection that guarantees that the fittest individuals in each generation are selected for reproduction. He produced an array, the size of the population, of individuals to choose from and selection was made from this array. Each individual has an expected frequency calculated and individuals with a frequency of n.something are included n times in the array. Once all individuals with an expected frequency greater than one have been included the remainder of the array is filled with individuals that have an expected frequency of less than one.

2.1.3 The Genetic Operators

2.1.3.1 Crossover

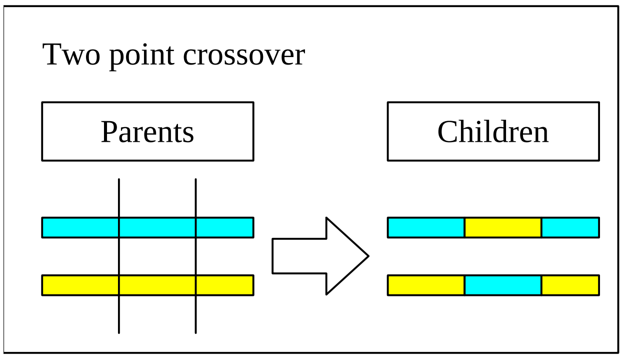
Crossover is the main genetic operator in most systems. It involves the recombination of two (or possibly more) parent chromosomes into one or two children chromosomes.

One point crossover works with two parents to produce two children. The effect of one point crossover is shown in figure 2.1-1. When using one point crossover genes nearer the middle are more likely to be separated than genes near the ends. One point crossover also requires some kind of inversion[5] for reordering of the chromosome to remove this potential problem



1 Figure 2.1-1: One point crossover

Two point crossover uses a similar technique with two crossover positions chosen. Two point crossover can be thought of as treating the chromosome as cyclic. Since the advantage inversion displays in one point crossover is no longer apparent, it is no longer required. The effect of two point crossover is shown in figure 2.1-2



2 Figure 2.1-2: Two point crossover

Uniform crossover is a gene-wise operator producing one child that assigns the child's nth gene from the first or second parent based on some measure of their relative fitness. Syswerda (1989) had greater success using a uniform crossover operator as opposed to using one or two point crossover on a series of functional optimisation experiments.

2.1.3.2 Mutation

Mutation must be used in extreme moderation as it is a dangerous and destructive operator. However it is required in any genetic system since it is the basis of introducing new genetic material into the population. Rechenberg (1965) and Schwefel (1965) both developed similar genetic techniques using only the mutation operator.
When using a binary alphabet for encoding, mutation requires only the flipping of a single bit.
When using a more complex encoding scheme, for example real value encoding, a common mutation operator is the addition of a Gaussian random number with mean

[5] Inversion is genetic operator that reverses the order of the chromosome between two randomly chosen points.

zero and standard deviation proportional to the individuals relative fitness. By using an adaptive mutation operator such as this the destructive effects on highly fit chromosome's is to a lesser degree then when it is applied to the more unfit chromosomes.
Another form of mutation with real value optimisation is the replacement of a position on the chromosome with a completely new random value.

2.1.4 Premature Convergence

Premature convergence is often apparent in evolutionary techniques due to the strong emphasis on crossover and the selection of the fittest individuals (Kursawe). Once convergence has occurred only the genetic operator of mutation makes changes to the population turning the search into a random walk. This is when a hill climbing heuristic can become useful to make use of both the strengths of a genetic technique and a gradient search based method.
Schraudolph and Belew (1992) used an approach they name *dynamic parameter encoding* as a means to avoid premature convergence. This technique uses a heuristic to determine when convergence has occurred and dynamically resizes the available range of each parameter to become smaller. This in effect "zooms in on solutions that are closer to the global optimum than provided by the initial precision" (Fogel, 1996, :95) If the global optimum is not included in the initial range of parameter values though this technique will be unable to find it. Schraudolph and Belew found that dynamic parameter encoding worked well when searching a quadratic bowl but poorly when searching a multimodal function such as Shekel's foxholes.

2.1.5 Specific Methods

There are three major forms of evolutionary techniques being genetic algorithms, evolutionary strategies (or evolutionary algorithms) and genetic programming (or evolutionary programming). A comparison of these techniques can also be found in (Fogel, 1993).

2.1.5.1 Genetic Algorithms

Formally a genetic algorithm uses only a binary alphabet to coincide with schemata theory. "Holland recognised that every evaluated string actually offers partial information about the expected fitness of all possible schemata in which that string

resides" (Fogel, 1996, :92-93, in reference to Holland, 1975, :66-74) This information, gained with many schemata, is termed 'implicit parallelism'.
Using a binary alphabet is powerful in the sense that the genetic operators working on chromosomes are quite simple. Mutation for example is simply the inversion of one position in the bit string. However the size of chromosomes for complex problems may be in the order of thousands of bits and can be slow and produce inaccurate values.

2.1.5.2 Evolutionary Programming

Rather than evolving specific solutions to a problem a collection of actual algorithms associated with the problem can be encoded and recombined. L. Fogel pioneered this general concept as a means of simulating evolution on a population of competing algorithms to develop artificial intelligence. (Fogel, 1962). He used it to evolve finite state machines for such tasks as predicting prime numbers (Fogel, 1966) and also with Burgin as a means of evolving strategies for simple games. (Fogel, Burgin, 1969) When evolutionary programming is applied to real valued optimisation problems they behave as evolutionary strategies, independently researched and described below.

2.1.5.3 Genetic Programming

One problem with using evolutionary techniques for neural network evolution is scaling. A fully connected network with N neurons will have N^2 connections and this produces impractical sizes for chromosomes. Genetic programming is a method of evolving a set of growth rules rather than a direct representation of the problem and can be thought as a *solution recipe*. This adds another layer of abstraction onto an encoding with chromosomes consisting of rules on how to build the actual phenotypic representation. Gruau (1994) developed an algorithm for compact cellular growth based on symbolic S-expressions as a means of creating network growth rules. Esparcia-Alcazar and Sharman (1995) found "Although this method can evolve very elaborate structures, we have observed that it takes very long to converge to an optimum, which is unsuitable for certain applications." (Alcazar, Sharman, 1995 :1)

2.1.5.4 Evolutionary Strategies

Evolutionary Strategies use a value type deemed necessary in the encoding of a chromosome. This is important in problems that use real valued parameters as a binary alphabet can not give the precision required without a long chromosome.

The evolutionary strategy approach was first explored independently by Rechenberg (1965) and Schwefel (1965) addressing the problem of real valued continuous function optimisation. “In this model, the components of a trial solution are viewed as behavioural traits of an individual, not as genes along a chromosome” (Fogel, 1996 :85)

Kursawe studied evolutionary strategies in the context of multiple criteria optimisation. To cope with the changing environment that is apparent with two or more criteria he employed the use of dominant and recessive genes in his encoding. His studies on co-optimising two complex functions showed exchanging the recessive and dominant genes for each individual with a probability of around 0.3 gave robust results. He concluded also from further testing that when only maximising one objective function the modelling of diploid[6] individuals was not worth the extra computation.

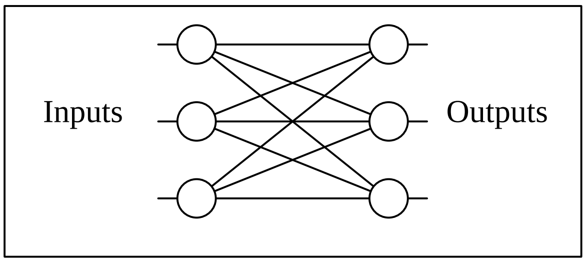
Two main approaches are in use today denoted by $(\mu+\lambda)$-evolutionary strategies and (μ,λ)-evolutionary strategies with μ indicating the number of parents and λ indicating the number of offspring per generation. In a $(\mu+\lambda)$ evolutionary strategy the μ fittest of all the solutions move into the next generation where as in a (μ,λ)-evolutionary strategy competition is only between the λ offspring with all parent’s being replaced.

2.2 Simple Neural Network Design

A neural network is a biologically inspired parallel-distributed processing method. It consists of a number of nodes (or *neurons*) connected by links. These nodes process the values on the links entering them by means of an activation function and distribute the result on the links leaving it. Each link has an associated weight that scales any signal passing along it and it is these weights that act as the network’s information storage mechanism. Teaching the network is usually achieved by manipulating these weight values. A number of nodes are reserved as the input and output of the network.

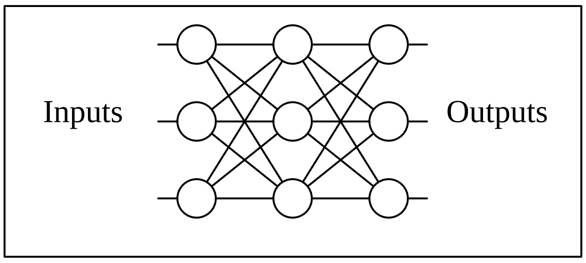
[6] Polyploidy refers to the number of distinct copies of the chromosome kept by each individual. Includes haploid (one copy), diploid (two copies), triploid (three copies) and tetraploid (four copies).

The simplest network is called a *perceptron* and consists of one layer of weighted connections. An example perceptron is shown in figure 2.2-1.



3 Figure 2.2-1: One level perceptron

A network can also consist of a number of hidden layers containing nodes not directly acting as either input or output. An example single hidden layer network is shown in figure 2.2-2



4 Figure 2.2-2: Single layer feed forward network

These networks are *fully connected* in that each node has a link to every node in the next layer. Such networks are also called *feed-forward* networks since links only exist from one layer to the next. *Recursive* networks can have connections from a node to any another node, regardless of the layer and can include links from a node back onto itself.

The activation function of each node takes the weighted inputs along all the links entering that node and applies some function to serve as the output for that node. This function is usually non-linear to produce a continuous response and needs to be differential if using a back propagation based training method. A sigmoidal function is often used since it produces a similar result to a simple threshold function but gives more accurate information when determining the error magnitudes that are needed for the training.

Training of a network is usually based on a gradient descent search of the error response surface called back propagation. Given the network's response to inputs and the actual desired result an error value can be calculated and fed backwards through the network to adjust weight values. For a more comprehensive discussion on back propagation algorithms see (Luger, StubbleField, 1995)

2.3 Evolving General Neural Networks

2.3.1 Overall Issues

The evolution of a neural network involves two parts, the selection of an appropriate network topology and the optimisation of the interconnecting weights. Both of these problems can be solved, separately or together, with a number of distinct approaches.

Issues that must be addressed with both stages of evolution include…

- How the representation of encoding scales to large networks.
- Whether reproduction operators create valid and more useful networks.
- Whether the best network can be represented by the encoding scheme.
- How invalid network designs are handled (usually left alone and subsequently ignored by genetic process due to the poor fitness values allocated to them)

There are two paradigms to designing a network's genetic encoding, low-level encoding[7] and high-level encoding[8]. Low level encodings are a specification of each connection and weight explicitly, and grow exponentially with the size of the required network. High level encodings encode a means of constructing the network (referred to as "growth rules" by Branke, 1995) and if encoded correctly are the same size regardless of the size of network's produced.

[7] Also known as strong or direct encoding.

[8] Also known as weak or indirect encoding.

An example of a combination of high and low level encoding is possible for example with space on the chromosome reserved for weight values (low level information) and also connection information (high level information).
When choosing an encoding scheme it is important to ensure human bias doesn't exclude networks that may be optimal.

2.3.2 Encoding

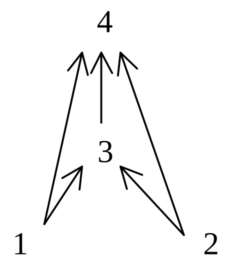
Real value encoding is one sensible choice for a low level network encoding scheme because it is more consistent and precise and results in faster execution (Michalewicz, 1992) (Thierens et al, 1993) (Yao, 1996).
Michalewicz (1992) also claimed that for extremely large state spaces, genetic algorithms perform poorly though "it is only fair to say maximising implicit parallelism will not always provide for optimum performance" (Fogel, 1996 :94)
Since weights are real values the use of binary encoding results in very large chromosomes with low precision and can slow down the evolution process. The simplest low level encoding for a network is concatenating all the network's weights into one string.
The main genetic operator crossover is more likely to separate gene information spaced apart on the chromosome so it is sensible to place similarly functional units close to each other. Thierens et al (1993) placed incoming and outgoing weights of a node next to each other.
Yoon et al (1994) placed all incoming weights of each node together and all nodes of each layer together.
Saha and Christensen (1994) used an encoding method that incorporates both weights and weight connections by supplying for each node an extra bit per weight representing whether that connection is present or not.
An example of an encoding used to describe the connection pattern of a possible network is shown in figure 2.3-1 (Miller et al, 1989)

Genotype representation: (0011001100010000)

Phenotype representation: Adjacency matrix

$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 0 & 1 & 1 \\ 2 & 0 & 0 & 1 & 1 \\ 3 & 0 & 0 & 0 & 1 \\ 4 & 0 & 0 & 0 & 0 \end{array}$$

Actual network connections



5 Figure 2.3-1: Connection pattern encoding

One major problem apparent in encoding is the permutation problem, (Yao, 1996) also referred to as the competing conventions problem, (Branke, 1995) due to the fact that many valid possible genotypes can map into one unique phenotype. “The group of functionally equivalent but structurally different networks can be defined by two simple transformations.” (Branke, 1995 :14)

The first is a permutation of the genotype that moves whole node information, leaving the phenotypic representation of the network unchanged. The second is the inversion of all the weights signs in a node with an odd activation function, again giving different representations of functionally equivalent nodes. (Branke, 1995).

There are also problems dealing with the consideration of the extra size of the state space (Branke, 1995) and the reproduction of unfit children using multiples of the one node. (Yao, 1996) With **n** nodes there are **n**! functionally equivalent nodes under the first transformation and 2^n under the second transformation.

Braun and Weisbrod (1993) attempted to avoid the permutation problem by making long connections less probable than short connections thus preferring the structured mapping with the shortest connection length.

Thierens et al. (1993) reordered the genetic string before applying crossover in such a way that nodes with a similar number of negative and positive weights are in the same general position on the chromosome.

2.3.3 Weight Optimisation

For a network with a fixed topology the selection of interconnecting weights is an optimisation problem with the goal to maximise the network's performance (Branke, 1995).

Evolutionary algorithms can be used for problems where gradient information is unavailable since they do not use it. This is apparent in problems for "networks with non differentiable transfer functions" (Branke, 1995 :4), recurrent networks and when using reinforcement learning methods (Yao, 1996). Also since they are a global search they can overcome many of the problems associated with local minima. However there needs to be a way of defining the performance of a network for the allocation of relative fitness.

With problems where gradient information is easily obtainable methods such as quickprop or cascade correlation usually outperform evolutionary approaches (Schaffer et al, 1992)

2.3.4 Topology Optimisation

> If the topology is too small (in terms of units and connections) the network might not be able to represent or even learn the desired input/output mapping. On the other hand, if it is too large, the network very often generalises poorly to inputs previously unseen.
> (Branke, 1995 :5)

There is no restriction on the topology of a network evolved by an evolutionary technique since they use no error signal back propagation (Branke, 1995). This can make evolutionary techniques appropriate for non-feed forward network designs such as recurrent networks.

2.3.5 Control Parameter Optimisation

The use of evolutionary techniques can also be applied to calculating control parameters for gradient based learning techniques. This can either be included as extra parameters in a hybrid approach or as its own genetic system.

Evolved parameters can include values such as learning rate, momentum values and the initial weight range. (Belew et al, 1989) (Marshall, Harrison, 1991)

Also work has been done on evolving parameters such as the activation function, bias values, the learning strategy, weight decay terms and the number of training epochs (Marshall, Harrison, 1991)

2.3.6 Fitness Evaluation

Fitness evaluation must take into account two factors, the performance and the size of the network. The most common performance measure is a function of the network's mean square error (MSE) in relation to a test set. Since a low MSE indicates good performance where as evolutionary techniques take low values as indicating a low fitness there needs to be some inversion mapping applied. Usually 1/MSE, 1/1+MSE or maxMSE-MSE (if a maximum MSE if known) are used for this mapping.
If a test set is unavailable, as is the case in such applications as robotic controllers, (Salama, Hingston, 1995) (Grefenstette, 1994) (Schultz, Grefenstette 1994) some measure of the network's performance at its given task needs to be defined.
As a means of selection Fogel et al. (1990) enforced that each network was only admitted to the next after competition with ten other individuals. The probability of a network 'winning' against another was equal to the opponent's error score divided by the sum of both error scores.
A heuristic can also be included in the evaluation of fitness to reward each good property of a network. Whitley et al (1990), for example, used a bias to allocate more of the overall training time to networks with a small number of hidden nodes.

2.3.7 Genetic Operators

When genetic operators such as crossover, inversion and mutation are to be used it has to be decided on what scale the operations act, or deciding "on what constitutes a gene" (Branke, 1995 :5)
Thierens et al (1993) developed a possible crossover operator that exchanged hidden node information with all incoming and outgoing weights.
In terms of crossover applied to the connection pattern of two networks Braun and Weisbrod (1993) allocated a connection to a child when both the parent's exhibit that connection. If only one parent had the connection then it is passed on with a user defined probability. The actual values of weights are also some user-defined function of the parent's weights.

"A good mutation operator should adhere to the principle of strong causality, ie. It should in most cases cause small differences in quality" (Utrecht, Trint, 1994)
To follow this causality Angeleline et al (1994) made all new connections created with associated weights set to zero and had new nodes added with no connections to other nodes.
Branke (1995) initialised new weights with small random values as well as removing low valued connections.
Fogel et al. (1990) used a mutation operator that added a random Gaussian number to selected weights and decreased the deviation of the variable over time as a means of annealing.

2.4 Evolving Neural Network Controllers

2.4.1 General Overview

Usually a controller for a robot system involves inputs coming from sensors of some type and outputs mapping onto a number of possible actions. Evolutionary techniques have been used successfully in a number of different problems to evolve controllers for robot systems. Usually though these systems are simulation based only and hence are different to real life models in many respects.
Grefenstette and Shultz make the comment that evolving a controller system "…will usually require that the learning system be given whatever level of knowledge can be easily provided by the designer." (Grefenstette, Schultz, 1994 :65)

2.4.2 Fitness evaluation

The evolution of robotic controllers presents an interesting problem in terms of fitness evaluation. Classifiers and networks used to predict time series usually use a function of the network's mean square error to determine fitness but since the behaviour of the controller is being evolved there is no unique numerical value that can be used for this. It is difficult to assign fitness for an individual since it will usually involve a certain amount of human bias and error.

2.4.3 Different Approaches

Wieland (1992) used a genetic algorithm to evolve recurrent networks for controlling a number of unstable systems including the broom handle balancing problem (also referred to as the inverted pendulum problem).

Lewis et al. (1992) evolved a network of fixed size and used their system to evolve the actual weight values for connections. They applied what they describe as *staged evolution* where different parts of the network are evolved separately. Their results show an improvement in the rate of global maximum convergence.

Cliff et al. (1993) used the SAGA genetic algorithm package to evolve controllers for a simple wheeled robot. The emphasis of their work was to design a structure that grew the size of the network for complex tasks and shrunk it for simpler problems.

Schultz and Grefenstette (1994) used a representation language approach to evolve simple robot behaviours. They define a behaviour as a set of ‘if..then..’ rules such as…

```
    IF front_sonar<30 AND bearing>10 THEN turn=20
    IF front_ir<5 THEN speed=-10
```

They also describe a system for including the rules in a hierarchical system as a means of higher level abstraction.

One of the major benefits with using such a representation is that “…it allows the learning system to be easily seeded with initial knowledge.” (Schultz, 1994 :2) Their initial population of solutions was generated as a combination of human generated rules and a number of variants on them. It seems though that using an approach of deriving initial individuals from human solutions may include some kind of human bias which may in turn inhibit performance.

Grefenstette (1994) talks about the same system and makes a number of points about using background knowledge. Constraints can be added to limit the generation of rules that are known to be undesirable though again this is introducing a form of human bias. If the allocation of fitness is correct than any undesirable rule sets will be removed by the evolution process itself.

Salama and Hingston (1995) used a matrix approach to store network connections in their system for evolving a robot controller where matrix element $a_{i,j}$ gives the weight from node *i* to node *j*. They evolved a simple robot controller for a 6 legged robot that walks to a target position using the minimum number of steps. An interesting concept in their project is the selection of mating pairs. They include a simulation of a finite grid that the networks inhabit. The networks are allowed to 'wander' randomly over the grid for a set amount of time, at the end of which they breed with the fittest network that they encountered. They believe this grid structure introduces a locality factor into the selection process that maintains diversity in the population.
Also included was a uniform distribution of noise applied to the position of the virtual robot to simulate some degree of real life noise. They conclude, "By inspection, it seems that a moderate level of noise during training is most beneficial." (Salama, Hingston, 1995 :582)

Maher and Poon (1995) propose an encoding method for general optimisation where the fitness function is encoded as part of the genotype and as such is co-evolved along with design solution. They believe this is an important part of many problems where the environment is changing and present a number of alterations to the standard genetic algorithm method.
Included is a design methodology for two-phase crossover, applied first to the problem part of the chromosome and secondly to the design solution part. They state "Optimisation is part of a design process, but it is not the whole. The design process includes the *search* for the problem as well as the solution." (Maher, Poon, 1995 :243)

2.4.4 Problems with Noise

One main concept in all controller based evolution is the handling of noisy systems. Normally simulation models do not accurately take noise into account and produce unrealistic results. Also the robustness of most controllers is an issue. Do the training procedures applied provide enough generality? Schultz (1994) believes that including more noise than is apparent in the real world environment makes evolved knowledge in a simulation model more robust.

2.5 Co-evolutionary Approaches to Control Algorithms

2.5.1 The Homogeneous and Heterogeneous approaches

There are two approaches to evolving a number of entities known as homogeneous[9] and heterogeneous[10] evolution.

Reynolds (1993) used a genetic programming approach to evolve "critters" that exhibited herding behaviour when attacked by predators. His system evolved a single homogeneous controller that moved each critter with information on its position, direction, neighbouring critters and predator locations.
Collins and Jefferson (1991) used a genetic programming approach to evolve a neural network controller for ants in an ant colony simulation. A homogeneous network controlled each ant with the fitness defined as the amount of food collected in a given time span. Inputs to the network included neighbouring information about food, pheromone and the nest. Outputs decided the movement of each ant and the laying of pheromone.
Haynes et al (1995) used a heterogeneous approach to a similar problem breeding teams of genetically programmed distinct individuals in a simple predator/prey system.
Koza (1992) developed a way of selecting heterogeneous individuals for a team at trial time that he termed *co-evolution*. A population under this scheme is divided into sub-populations with each one providing a specialised member for the team.

Whether a problem is homogeneous or heterogeneous makes a large difference in the breeding policy of the algorithm. In the homogeneous approach each member of the population is evolved as normal and a team is constructed by 'cloning' this individual. In the heterogeneous approach there is the decision whether members should be allowed to breed only with other team members or whether they are allowed to breed between teams. (Luke, Spector, 1996)
An unfortunate problem associated with constructing teams in this way is the so-called *credit assignment problem*. When a team of entities has had a fitness value

[9] A common algorithm control different entities.
[10] Distinct algorithms control different entities.

evaluated for them as a whole, which individuals get more credit for the teams success or failure? (Haynes et al. 1995)

Haynes et al. addressed this problem by considering the whole team as one individual. There are facilities for defining sub-individuals within a main individual constructed by Koza (1994) *Automatically defined functions* and Spector (1996) *Automatically defined macros.*

2.5.2 Communication Systems and Coevolution examples

Luke and Spector (1996) in their research developed a simple predator/prey environment with one ‘gazelle’ and a number of ‘lions’, the aim of the lions being to catch the faster gazelle. They tested a number of different approaches to team selection for the predators and also a number of predator communication systems. Sample runs found that restricted breeding between team members outperformed[11] free interbreeding for predators that had distinct control algorithms. They also addressed the problem of deciding a means of communication between members of each team. Their sensing experiments compared name-based sensing[12] and deictic sensing[13] and found that the former outperformed the latter in all cases considered. They also found that “…as the sensing becomes increasingly distinct (more name-based), heterogeneous approaches work better than homogeneous approaches.” (Luke, Spector, 1996 :2)
Naghashi et al. (1995) review a number of approaches to evolving neural controllers but point out common problems apparent in most. The first is that when using a homogeneous approach all entities act in the same manner when presented with the same conditions. This means there is no unique learning mechanism in the scope of one entities lifespan, each simulation gives only an evaluation of how the controller performs and does not act directly towards improving it. They address this problem by giving each entity it’s own independent learning mechanism modelled with what they call *a classifier system*. The simulation used genetic programming to evolve

[11] In terms of the speed of evolution.

[12] Where entities are referred to explicitly (eg ‘entity number 5’)

[13] Where entities are referred to implicitly (eg ‘nearest neighbour’)

"if..then.." rule structures for controlling the entities and observed the evolution of mutualism between different entities as a means of survival.

The simulation consisted of three distinct entity types A, B and C with the following main characteristics.

1. A preyed on B and B preyed on C but there was no interaction between A and C.
2. Each of the entities had a means of sensing the proximity of others.

If an entity caught another it gained that entities strength and this was used as a measure of the fitness of each individual. Their research found that C evolved a behaviour to stay close to A to avoid being caught by B.

3 Design and methodology

3.1 Network encoding

3.1.1 General comments.

The genetic system defined for this project evolves the interconnecting weights and bias terms for the nodes in the single hidden layer while also performing limited optimisation of this layer's topology. Although this limits the behaviours of the networks that can be defined, it gives rise to a simple encoding scheme.

As discussed previously in section 2.3.2 the classic genetic algorithm approach of using binary encoding does not seem appropriate for neural networks so real based encoding was used. For a neural network encoding this is sensible since any fully connected network can be defined by a sequence of real values (representing the connection weights and bias values).

For this encoding scheme the number of input and output nodes is fixed and to make the DNA a fixed length a maximum number of hidden nodes is set prior to any program execution.

3.1.2 Encoding scheme.

The encoding of a network is made up of a sequence of genes each representing a potential node in the hidden layer. In turn each gene is made up of three sub parts, the active position, the incoming weights and the outgoing weights.

3.1.2.1 The active position

The active position is a single real value that determines whether the hidden node represented by the gene will be exhibited in the final network. This is a form of high level encoding with a non-negative value meaning this node will be present in the network and a negative value meaning this node will not be exhibited in the network. This is a very rough model of diploid behaviour in genetics where a section of DNA can be 'turned off' with the possibility that it will become useful in the future and 'turned back on'. The changing of the active position through mutation is the systems way of performing topology optimisation. Though this value could have been represented in the gene string as a single bit, for the simplicity of the implementation it was assigned to a real value so that the whole chromosome consisted of only real values.

3.1.2.2 The incoming weights

The next section of the gene represents the incoming weights. The first of the values is the bias value for the node with the remaining values being the actual connection weights from the input layer to the hidden layer. The bias value was included with the incoming weights due to the implementation of how the bias values are used. This section of the gene is a low-level encoding where the values in the gene itself map directly onto the connection weight values.

3.1.2.3 The outgoing weights

The remaining values in each gene represent the weights of the connections from the hidden layer to the output layer. This section, as with the incoming weights, is a low-level encoding where the DNA values are mapped directly to the connection weight values.

3.1.2.4 Encoding scheme details

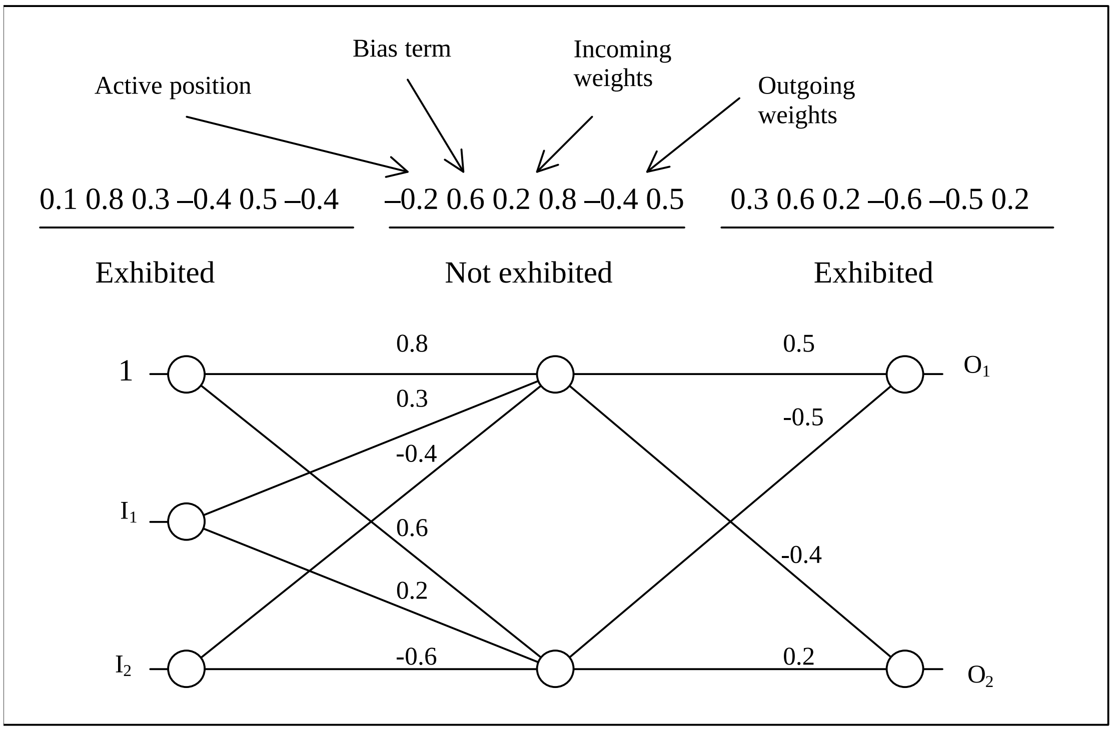
For the general case of a network with I input nodes, a maximum of H hidden nodes and O output nodes the chromosome for a single network is encoded in a string of H(1+I+O) real number values (with the 1 representing the active position).

It can be argued that the positioning of these three groups is important. For instance with the active position next to the inputs there seems to be more of a chance of the active position and inputs being passed to a child than the active position along with the outputs. This problem is irrelevant though with the use of a cyclic crossover operator so that the active position, inputs and outputs are effectively all neighbours to each other. This would not be the case with 4 or more distinct groups of information being represented by each gene.

3.1.2.5 A small example

The encoding for an example 2x3x2 network (2 inputs, a maximum of 3 hidden nodes and 2 outputs) is as shown in figure 3.1-1. With 2 functional inputs there are actually

3 inputs to the network with the first hard-wired to the value 1. The connection weights from this input to the hidden nodes represent the bias values thus giving a functionally equivalent implementation of how a bias value acts in a standard network.



6 Figure 3.1-1: An example of encoding for a 2x3x2 network.

3.1.2.6 Other aspects of the encoding

If the DNA for a potential network is initialised with random values then approximately half of the genes will have a non-negative value in the active position. This represents a network with only half the hidden nodes active and hence exhibited. Therefore if it desired that the network has H hidden nodes then the actual DNA should be defined to have a maximum number of hidden nodes equal to Hx2. In this way the average number of hidden nodes exhibited by each network initially will be H.

The encoding scheme as it is uses a large degree of low level encoding. This method encodes weight values accurately but does not scale well to large networks where doubling the size of a network effectively doubles the size of the encoding.

The system also only defines fully connected networks with each hidden node connected to every input and output node. The system can effectively simulate a non-connection by having zero-weighted values but this is not reflected in any fitness functions applied during simulation and as such is not treated as a benefit in any explicit way.

3.2 Fitness Evaluation

3.2.1 Simulation methods

When evolving networks the only way to get accurate fitness values is to express the DNA in its phenotypic network form and run it through the task. This presents the major bottleneck in the simulation where a single simulation run may take some time. When the entire population must be simulated the program spends a large proportion of its time in simulating to gain fitness values.

One approach to dealing with this bottleneck is annealing the simulation length. For example if the task is expected to take **n** turns in the simulation model we start with the simulation lasting $\mathbf{n_0} < \mathbf{n}$. and each epoch we increase $\mathbf{n_0}$ until it reaches **n** where the whole simulation will be performed. This gives an obvious speed up as the less time is spent simulating though it has some major drawbacks. Without a full simulation being applied members have an inaccurate fitness assigned to them. Individuals that are good at the first parts of the problem dominate the population too much in the early stages and the system has trouble learning the final stages of the task. Annealing was tried as an approach in the path learning example (see section 4.3.1)

Another problem with having to perform simulation is dealing with any random factors that may be present in the initial set up of the simulation model. An example of this is assigning random starting positions to entities. An entity tested with a ‘good’ starting position will usually get a better fitness than one with a ‘bad’ starting position even though they may have performed equally if given the same position to start with. The simplest and most effective approach to solving this problem is to perform multiple simulation runs and take some form of average. This of course takes much longer but allocates fitness fairer, though it still does not guarantee a fair trial for each individual.

3.2.2 Fitness rescaling

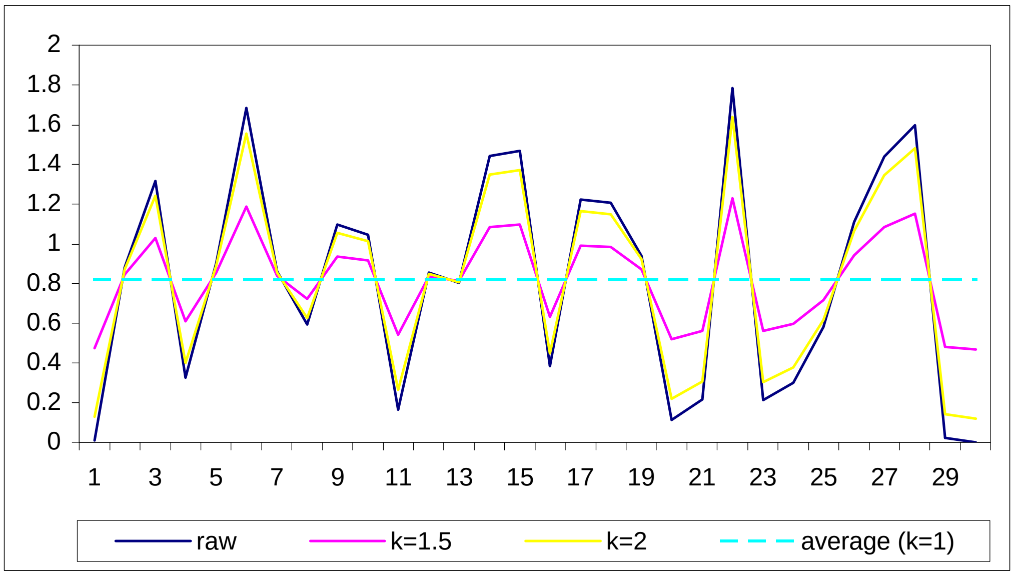
Recall that fitness rescaling is converting the raw fitness values obtained through simulation to the scaled fitness values to be used in the selection process.
There is a need for rescaling to deal with two diversity problems apparent in any genetic system.

Firstly when evolution starts there are usually a few ‘lucky’ individuals whose DNA give them large fitness values compared to the others, even though they are not the ideal individuals. These extreme values often mean these members swamp the population in only a few epochs resulting in premature convergence.

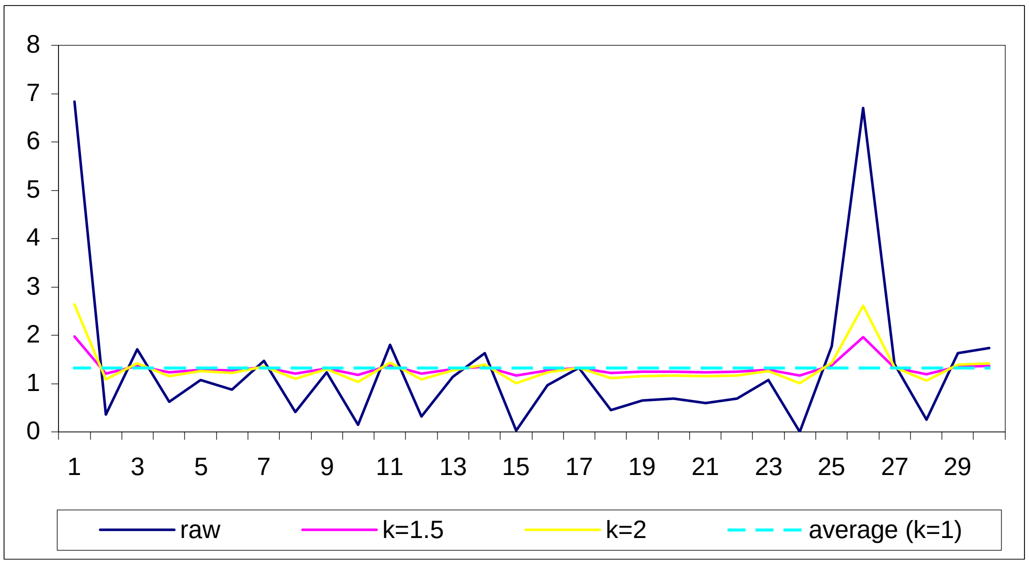
The second problem is when the system is converging on an optimal solution and the population consists of only high fitness valued individuals. In this case the system cannot accurately select the fitter members over the others and some rescaling is needed so that selection can properly determine the best individuals.

Best results were gained using the ‘scale maximum relative to average approach’ (see section 2.1.2). Rescaling to make the maximum twice the average was found to give robust results agreeing with Goldberg (1989). Since this value implicitly determines the convergence rate of the system a lower value (towards 1.5) with a large population maintained reasonable diversity. Values too close to 1, indeed 1 itself, treat all members equally and hence are useless.

Figures 3.2-1,3.2-2 show examples of this scaling using a number of different values for **k**. It is notable that this technique can map some fitnesses to negative values. Since these negative values will upset the standard roulette selection the algorithm needs to deal with these negative values. All values can have a constant added to them so the most negative value is scaled to zero or the algorithm can simply set the negative values to zero. The former is preferred since setting all negatives to zero treats these members unequally.



7 **Figure 3.2-1: The effect of different scaling values on similar values**



8 **Figure 3.2-2: The effect of different scaling values with distinct peaks**

3.3 Selection

Standard roulette selection was used in combination with population elitism. This produced a selection system that correctly selected proportional to fitness while guaranteeing that the best solution was maintained. For a population of N members the selection function was called at least N times. (Possibly more since in the case of crossover another parent needs to be chosen)

3.3.1 Naive roulette selection algorithm

The naive approach to implementing roulette selection requires that only each fitness value and the sum of the fitness values is known.

An individual is selected as follows…

```
1. Select a random number, n,  from 0 to the fitness sum
2. j = 0
3. if n ≤ fitness[j] then the jth individual is selected and we are
   finished
4. else n = n-fitness[j] and j=j+1
5. go to 3.
```

3.3.2 Improved guessing roulette selection algorithm

A guessing approach developed requires a selection array constructed in the following way…

```
select[0] = fitness[0]
select[j] = fitness[j] + select[j-1]  1≤j≤N
```

The selected individual will be member j where select[j-1] < n ≤ select[j] (except for the boundary case of n ≤ select[0]), all that is needed is to find the correct value of j. This is done by making a guess of what j should be and refining the guess

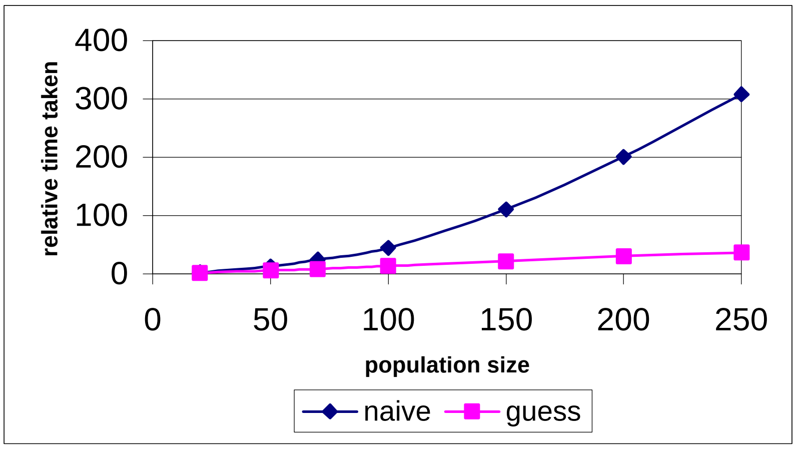
This time the algorithm is as follows

```
1. select a random number, n, from 0 to fitness total
2. if n < select[0] we choose member 0    //lower boundary condition
3. if n > select[N-1] we choose member N  //upper boundary condition
4. guess j = n / average fitness value
5. if select[j-1] < n ≤ select[j] we choose member j and finish
6. if n > select[j] then j=j+1 and go to 5.
7. j = j-1 and go to 5.
```

This approach has the advantage of a great deal of speed up. Even though the select array must be created it needs only be done once for each epoch where as the actual selection will be performed N times.

3.3.3 A comparison

Figure 3.3-1 shows the time taken for each selection method executed a thousand times on a population with a thousand individuals.



9 Figure 3.3-1: A comparison of the times needed for different selection techniques

It can be seen the naive approach is of order (n^2) where as the guessing technique performs better with a performance of order (n)

3.3.4 Elitism and Pair-wise elitism

Since selection is still an essentially random process it is possible that the fittest individual in any population may be lost simply by not being chosen. Recall that elitism is the act of taking the best member of a population and copying it without changes to the next generation. Without some form of elitism the population is not guaranteed to converge, either on an optimal solution or otherwise.

Pair-wise elitism is a further refinement developed for this project to further direct convergence. In the pair-wise system a new member is created as normal by both selection and the genetic operators to fill each position in the new generation. If the fitness of the new individual is less than the member who previously occupied that position then the new member is discarded and the previous member is replaced. Pairwise elitism was shown to give good diversity and slower premature convergence on some types of problems (see section 4.3.2)

The pairwise approach also guarantees that the average fitness of the overall population will convergence on the elite fitness. Since both of these elitism techniques have potential side effects on the evolutionary system they have associated with them probabilities that decide whether they are applied each epoch. (Eg. If the elitism probability defined is 0.6 then elitism will be performed during 60% of the generations)

3.4 Selective genetic operators

3.4.1 General comments

The three genetic operators implemented were crossover, mutation and inversion. Since the encoding has such a specialised form it was decided that these operators should be tailored for this specific encoding.

The nature of the genetic methodology does not require this to be the case though all information that is general to the problem domain and does not bias solutions helps speed the evolution.

These genetic operators define each position in the chromosome to be in of one of several categories. When choosing a position within the chromosome (eg. a location for crossover when performing crossover) each category has an associated probability assigned so that some places are more likely to be chosen than others.

3.4.2 Weighted values for crossover position selection

Intuitively it was decided that there are two main ways of recombining two networks to construct a new network using crossover.

The first is to exchange hidden nodes and this is reflected by defining a probability for the position between genes on the chromosome.[14]

The second is to exchange incoming and outgoing weights between hidden nodes. This is reflected again by defining a probability for the gene position where the output values start.

[14] ie Between hidden node information on the DNA

Finally of course there must be a chance that a crossover point can occur anywhere since this is the strength of the underlying evolutionary principle of crossover. For the simplicity of the implementation it was coded so that in the third case it was possible to choose a position from case one (between nodes) or case two (between incoming and outgoing weight information) making the three possibilities not mutually exclusive.

3.4.2.1 An example of possible crossover positions

In the case of a network with 2 inputs, 3 hidden nodes and 3 outputs a chromosome is of the form `ABIIOOOABIIOOOABIIOOO`[15]. Figure 3.4-1 shows the three possible positions then for crossover.

```
Case 1: crossover location between nodes
  A B I I O O O|A B I I O O O|A B I I O O O

Case 2: crossover location between incoming and outgoing weights
  A B I I|O O O A B I I|O O O A B I I|O O O

Case 3: crossover location anywhere
  A|B|I|I|O|O|O|A|B|I|I|O|O|O|A|B|I|I|O|O|O
```

10 Figure 3.4-1: Possible choices of the location of a crossover point

Each of the three positions has associated with it a relative probability say C_{node}, C_{weight} and $C_{anywhere}$. If C_{total} is defined as $C_{node} + C_{weight} + C_{anywhere}$ then the probabilities of each occuring is C_{node}/C_{total}, C_{weight}/C_{total} and $C_{anywhere}/C_{total}$ respectively. Once one type has been chosen by these probabilities it is used to determine the offset within a randomly chosen gene on the chromosome.

[15] A-node active position, B-bias term, I-incoming weight, O-outgoing weight

3.4.3 Weighted values for mutation position selection

Weighted probabilities were also assigned to different positions on the chromosome for the choice of the position where mutation could occur. Again these were chosen to reflect the different attributes that the positions represent. Two types of position were chosen; firstly the active position and secondly the positions of weight values (including the bias term). Since the changing of an active position effects the network much more than the changing of a weight the relative probability of active position mutation was assigned much lower than that of a weight value. The algorithm for deciding where mutation occurs is the same as that used in the case of crossover. First a total is calculated and used to determine which type of position the mutation will occur at (active or weight position). Once this is determined it is used to calculate the offset in a random gene of the chromosome.

3.4.4 Inversion positions

Inversion was applied on the scope of whole nodes so that the functionality of the network remained the same. It was believed that inversion at the level of an individual connection weight would be too destructive and hence serve no purpose. Inversion even though it brings out problems dealing with competing conventions was still included to increase diversity and allow the possibility of evolving networks with multiple instances of the same hidden node. Inversion was assigned the lowest probability of occurring so that some stability was retained in terms of a nodes position in the chromosome.

3.5 Population Management

3.5.1 The concept of sub-populations and migration

The problem of pre-mature convergence with any evolutionary strategy is reduced by somehow maintaining population diversity. One means of maintaining this diversity is to split each population up into a number of distinct sub-populations so that a dominant individual in a sub population can not effect the whole population. However if these sub-populations are keep distinct then the global search power of the system is lost. Migration is the act of moving a number of individuals between the sub-populations

3.5.2 Migration implementation

Migration is performed with a predefined constant frequency. If migration occurs too frequently then the diversity of keeping sub-populations is lost. On the other hand if it occurs not frequently enough then each sub-population will converge on separate values. The number of individuals involved in a migration is a predefined constant, usually a small fraction of the sub-population size.

Each time migration occurs a number of individuals are chosen from each sub-population to be moved to the next[16] sub-population.

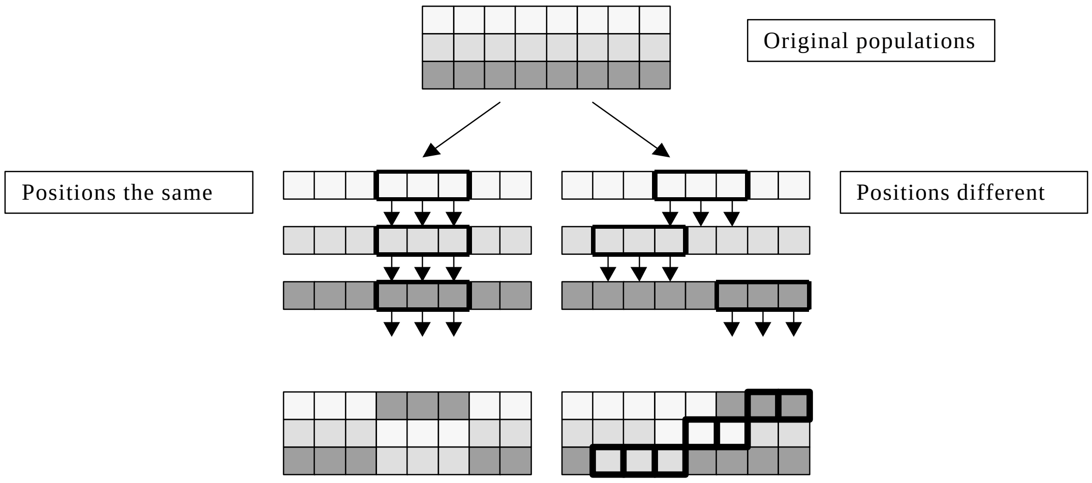
The individuals that are chosen for migration must be located in the same position in the sub-populations so that members are not lost. This concept is best explained with the following two examples.

Consider a system of 24 individuals with 3 sub-populations, each consisting of 8 members and a migration size of 3 individuals. Figure 3.5-1 describes the two possibilities for choosing the members who will be migrated.

The left-hand side shows the result from choosing a single random section of the population and performing the migration with each sub-population using this single section. As can be seen all members are retained with those in present in the section change their sub-population membership.

The right-hand side describes the result of choosing a unique section for migration in each sub-population. In this case some members are lost when others are copied over them (denoted by bold lining in the final set of the right hand side). In the same way some members under this scheme have a second copy of themselves created in the population. Since this loss of members occurs randomly, regardless of the fitness values of the members involved, it should be avoided and the left hand approach adopted.

[16] Cyclically next



11 Figure 3.5-1: Migrating with and without a single subsection

3.5.3 *Introducing completely random members*

To further maintain diversity in each sub-population a completely new random individual can be introduced each epoch with a predefined percentage chance. To cause the minimum disruption to the population this new individual should take the place of the member with the lowest fitness. This concept can be thought of roughly as a mutation operator working on the whole population since the role of a mutation operator is to introduce new material into the system.

3.6 Incorporating Back Propagation

3.6.1 *How it can be useful*

As previously stated back propagation is useful as a fine tuning technique when the genetic system has converged. It can also be useful near the start of a simulation to direct the evolution towards a desired type of solution. Such usage though introduces human bias into the system which should be avoided whenever possible.

3.6.2 *Why back propagation was avoided*

Back propagation is only useful when exact solutions are already known. Since one point of this project was to apply evolutionary techniques where such exact solutions are unknown it was avoided, even though it has strengths when dealing with neural network architectures.

3.6.3 Benefits of not using back propagation

Back propagation uses differentiation of the transfer function to determine error magnitude information. It is thus required that the transfer function is continuous. If back propagation is not to be used then a continuous transfer function is not required. Simpler functions such as a standard step function can then be used. Of course back propagation is applicable only to feed forward architectures and by avoiding it completely more freedom is allowed when evolving the topology.

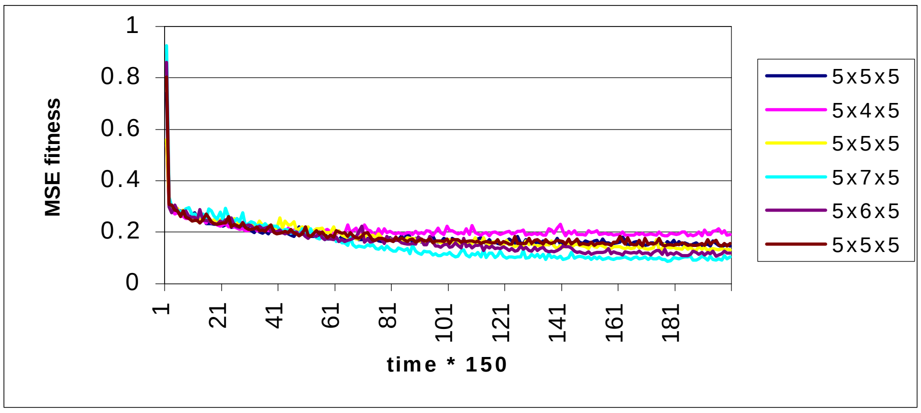
4 Results without communication

As the code was developed a number of test cases were implemented. All cases included evolving neural networks with the obtained results giving feedback for further refinement of the algorithms and code implementation. The first cases described in this section tested the details of the problems focusing on using one population and evolving for tasks requiring only single entities. Communication results are detailed in the next chapter.

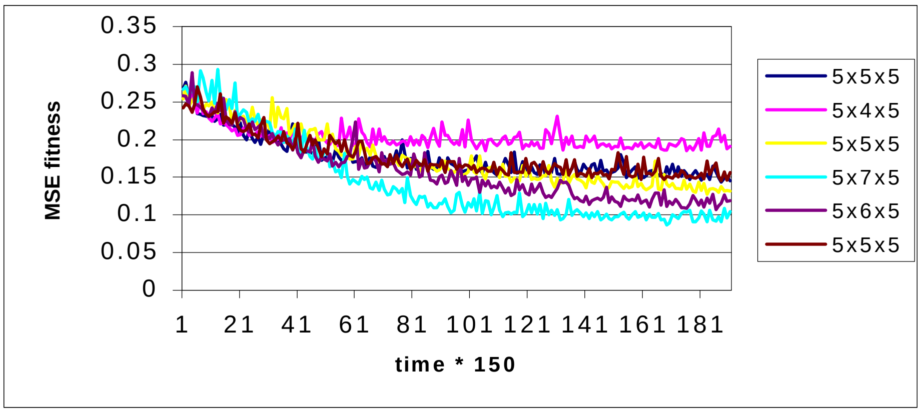
4.1 *Data prediction*

The simplest and most common application of neural networks is the learning of a simple data set. With the initial framework prepared for the network architecture a test case of learning a random data series was trialed.

A data set of 10 elements was randomly defined associating 5 random inputs with 5 random outputs where these values varied from +1 to -1. A single population was maintained with a crossover probability of 0.7 and mutation rate of 0.01. Inversion at this stage had not been implemented. Fitness was first defined as the sum of the mean square error. Figure 4.1-1 shows the results from a evolution of 30,000 epoches. Since the system makes a great improvement in the first few hundred epoches figure 4.1-2 shows the same graph from epoch 3000 to epoch 30000 to increase clarity.



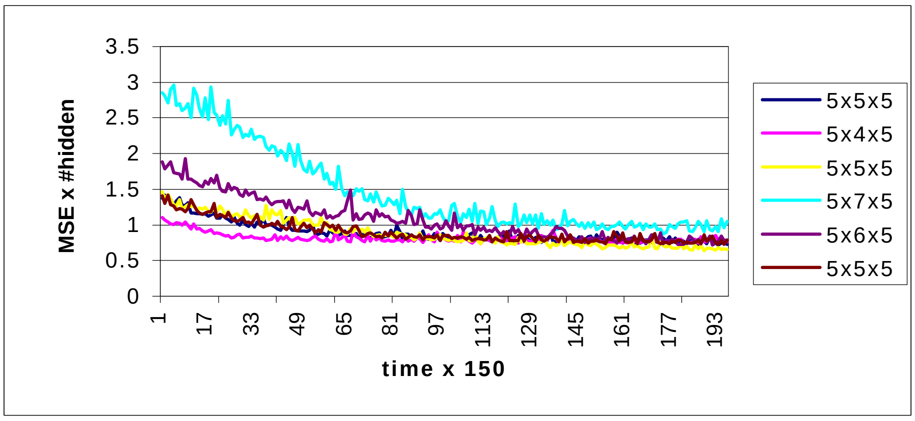
12 Figure 4.1-1: Evolving data series prediction, MSE fitness 1



13 Figure 4.1-2: Evolving data series prediction, MSE fitness 2

In can be seen that, as expected, the networks with the most hidden nodes performed the best.

Next the fitness was defined as the mean square error result multiplied by the MSE by the number of hidden nodes exhibited in the network. Results such as figure 4.1-3, again showing only epoches 3000 to 30000, show the rewarding of networks with less nodes though they perform the worse.



14 Figure 4.1-3: Evolving data series prediction, MSE x #hidden fitness

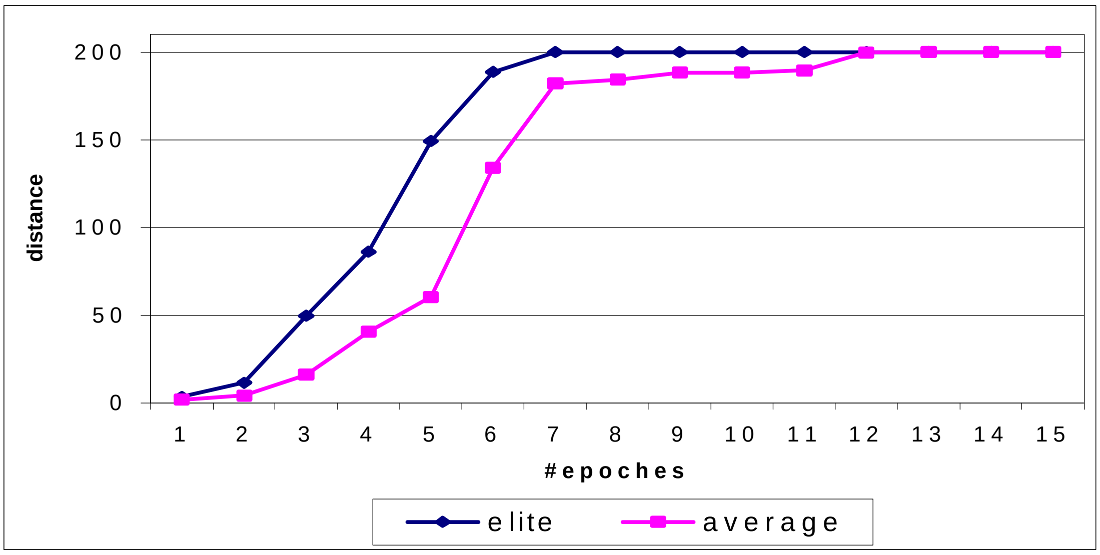
Even with this ability to somewhat define a need for a small number of nodes the evolutionary system could not improve much past a mean square error of 0.1. With only 10 values in the data series varying between +1 and –1 this is not a very accurate result. This is an example where the evolutionary system has been able to quickly give an approximate solution without being able to fine tune.

4.2 *Straight line movement*

The first test case of an actual controller was the evolution of a simple straight-line walker. A single entity controller was evolved in a single population for the task of maximising the distance travelled in 200 turns.

As inputs the controller received a series of random vectors of length five. The values were random but predetermined so that on each execution of a simulation an entity would receive the same data series.

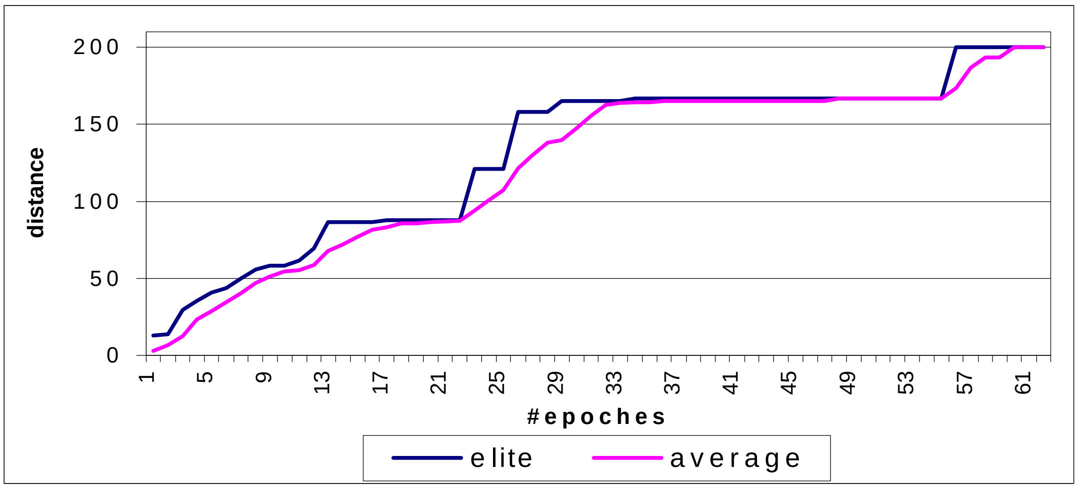
With outputs being a decision to turn left, turn right or walk forward it is simple to determine the perfect entity for this task is one that always walks forward. The fitness function was simply then the distance travelled after the 200 turns.



15 Figure 4.2-1: Evolution of a simple straight-line walker

Figure 4.2-1 shows the average values of three simulation executions. A population size of five was used with a mutation rate of 0.01 and crossover rate of 0.7. A maximum number of four hidden nodes were used. Again by this stage inversion was has still not be implemented.

Since it is desirable to reward a low number of hidden nodes the simulation was further refined to have the fitness values divided by the number of exhibited hidden nodes. Figure 4.2-2 shows the average of three runs with the same values for population size and mutation and crossover rates. For each population a perfect walker was evolved before epoch 20 though it took longer to further evolve the networks to use a lower number of hidden nodes.



16 Figure 4.2-2: Evolution of a simple straight-line walker rewarding fewer hidden nodes

Two plateaus are apparent in the graph. The first is from epoch 13 to epoch 23 where two of the three runs have evolved a perfect walker with 2 hidden nodes. The third run took longer to evolve to this degree and as a result this plateau is just under 100[17]. The second longer plateau from epoch 30 to epoch 53 is the result of two of the runs evolving the perfect walker using only one hidden node. All three had finally evolved the perfect walker by epoch 60 with only one hidden node.

This example though it shows good results in terms of the evolutionary system, is somewhat contrived in terms of the networks. The solution for this problem is to always walk forwards, ie give a high output on the forward output node and low values on the turn left and turn right output nodes. Quite often with a random weights defined for a network one hidden node will dominate all the other nodes in the layer

[17] 100 being the perfect score for a network with two hidden nodes.

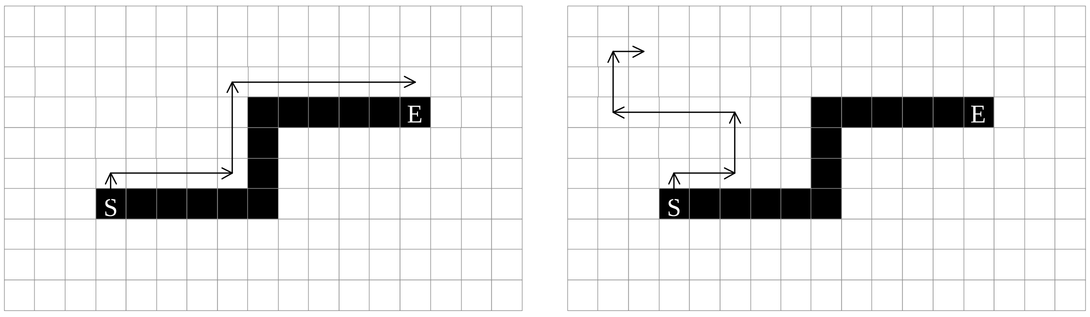
resulting in one single output node firing constantly. This was apparent with one simulation run where a perfect walker was present in the first random population

4.3 Path navigation

4.3.1 Navigating a simple path

For a more complex version of evolving a walker the problem was changed to an entity having to navigate along a simple path. This time the inputs were the coordinates of the entity (x and y) and the direction faced (in discrete multiplies of 90 degrees). Outputs defined the decision to turn left, turn right and to move straight ahead. The entities had no direct knowledge of the path with fitness calculated as a function of the amount of path covered in a given fixed amount of time. The path can be thought of as being on a black and white grid with the black squares defining the path.

The first simplest fitness function tested was defined as rewarding one point towards the fitness for each black square traversed. To ensure that the same piece of the path can not be counted twice every time a black square was covered it was changed to become white.

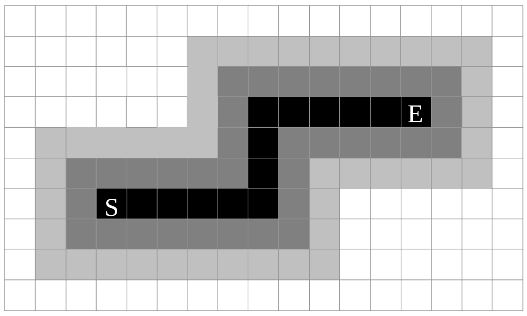


17 Figure 4.3-1: A good and bad attempt at navigating a simple path

This fitness function works but is rather discontinuous. For example the two possible walks shown for a simple path in figure 4.3-1[18] are both allocated the same fitness under this function even though the first is obviously a better solution.

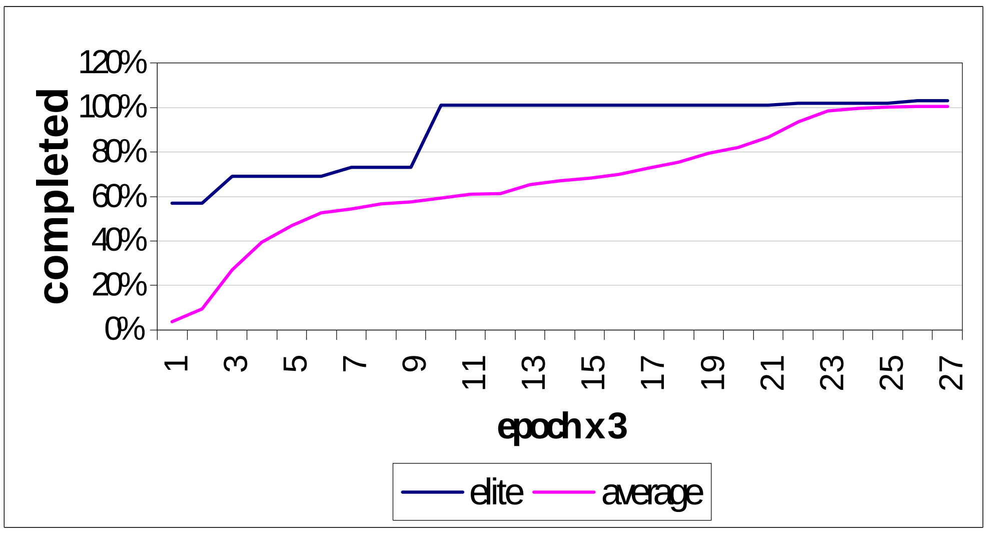
[18] S-start of path, E-end of path

This presents a good example of being able to put prior knowledge of the problem into the evolutionary system, in this case to produce a more continuous fitness function. By grey scaling the grid with extra grey squares that represent partial fitness points the fitness function can more accurately rate potential solutions. The path can then be of the form shown in figure 4.3-2. Now fitness is assigned by allocating points based on how dark the square covered is. This concept can even be extended to define outer regions to represent negative values so that going in the wrong direction can be penalised.



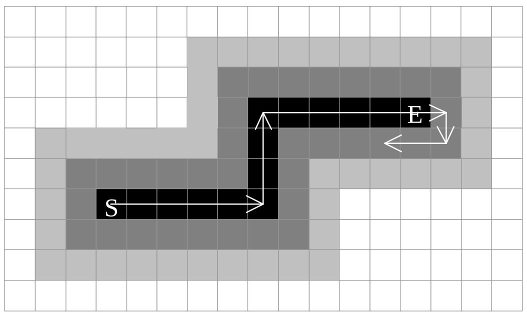
18 Figure 4.3-2: Grey scaling the path for a more continuos fitness function

The evolutionary system was able to evolve a solution for traversing the path as well as bringing out an interesting trait in the fitness function that had been previously unthought of. Figure 4.3-3 shows the elite and average member information for a single evolved population of 30 members.



19 Figure 4.3-3: The evolution of a simple path follower

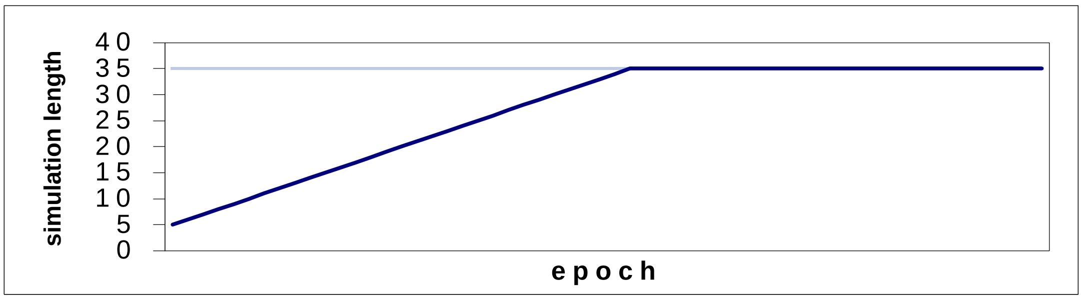
A perfect navigator is evolved by epoch 30 but it can be noted that an individual was eventually evolved that gained more than 100% on a single trial. This is a result of the fixed amount of turns given to traverse the path being more than what was required. This combined with the grey scaling that was applied for a smoother fitness function gave the opportunity for some individuals to slightly 'cheat' as shown in figure 4.3-4. A good example of greedy optimisation.



20 Figure 4.3-4: Path followed by the elite member evolved

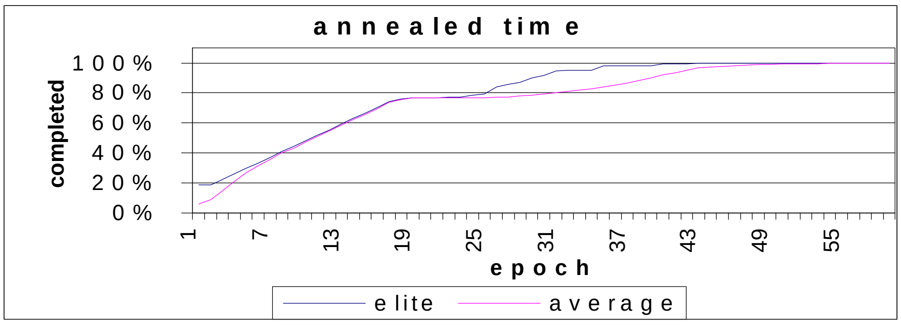
The number of hidden nodes evolved for this simple path averaged at just over four, while the entities that solved the path normally usually had two hidden nodes. These figures roughly correspond to the number of turns needed for completing the path.

Annealing of the simulation length was then tested with the relationship between the simulation run length and the current epoch described by figure 4.3-5. Note that the length flattens at 16, the time needed to traverse the entire path.

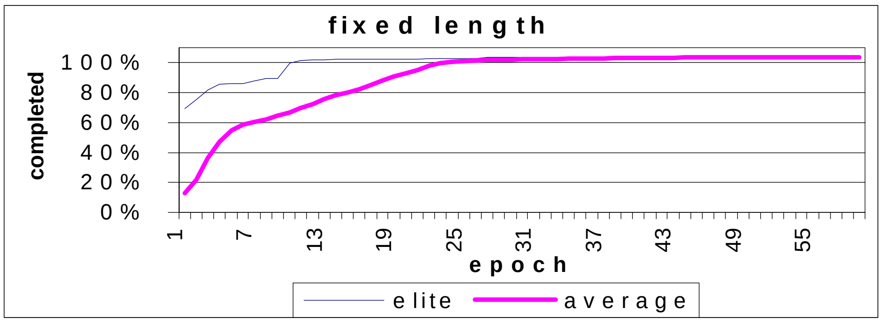


21 Figure 4.3-5: The relationship between epoch number and simulation length

Figures 4.3-6 and 4.3-7 show the comparisons between using this annealing technique and allowing each epoch to run for a full term. Again these graphs represent the average of three complete program executions. The annealing approach took approximately 40% of the time to execute though it took longer to evolve with both trials eventually evolved a perfect member. In can be seen though that all diversity was lost in the annealed case and as such the simulation relied only on mutation to better the population.



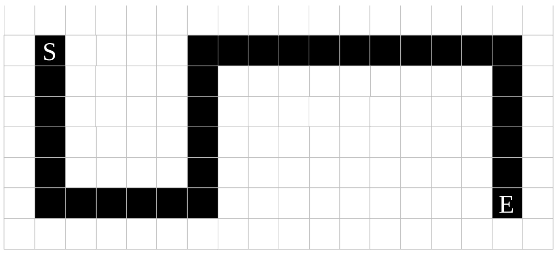
22 Figure 4.3-6: Evolution of a path follower with annealed simulation time



23 Figure 4.3-7: Evolution of a path follower with fixed simulation time

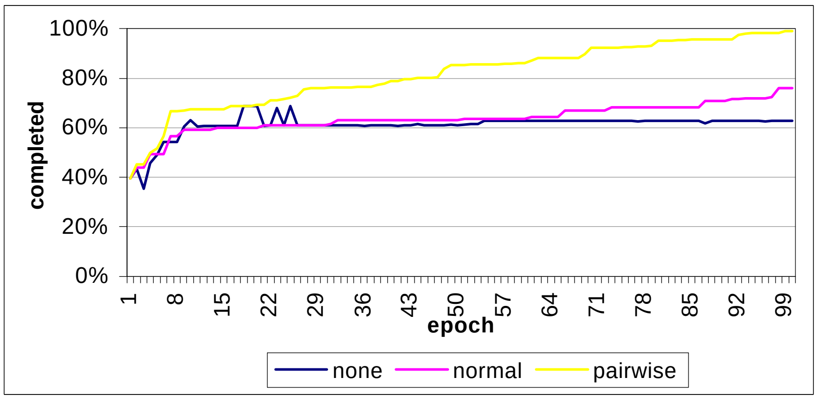
4.3.2 Navigating a more complex path

As a further test of the evolutionary system the path was extended to a more complex design and the facility for pairwise elitism was implemented. Figure 4.3-8 shows the more complex path used for testing without showing the grey scaled smoothing that was used as before. A single population of thirty members was maintained with crossover and mutation probabilities defined as for the simpler path example.



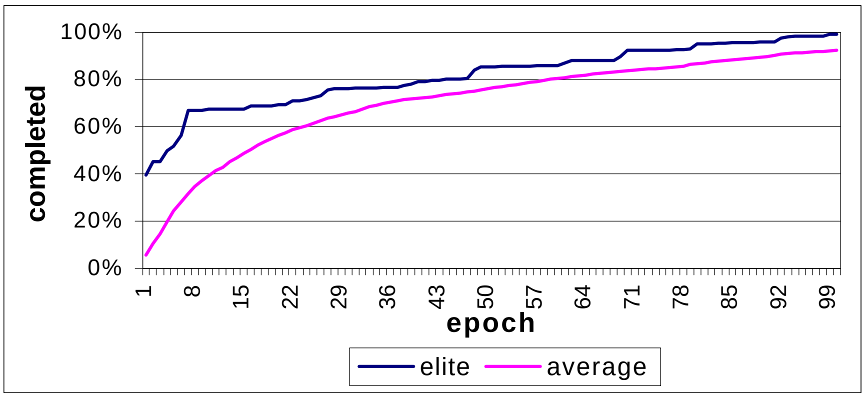
24 Figure 4.3-8: A more complex path

Figure 4.3-9 shows the average evolution of the elite member from three runs using no elitism, normal elitism and pairwise elitism. The pairwise approach performed well in this task converging faster on the optimal path navigator.



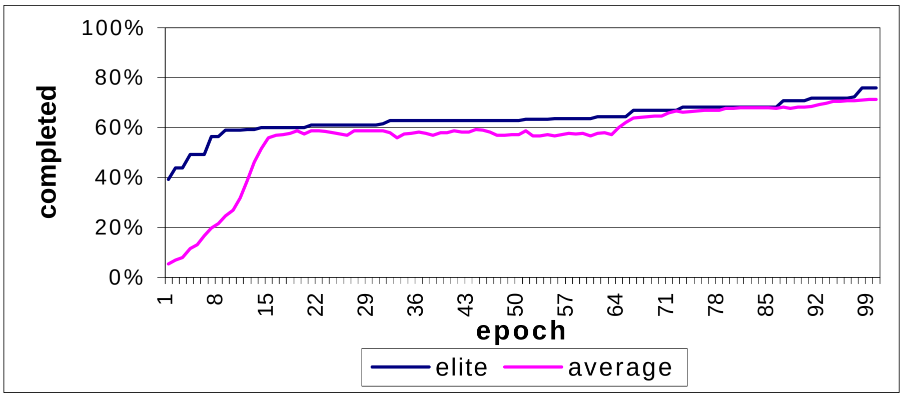
25 Figure 4.3-9: Comparisons of different elitism techniques with the complex path

The pairwise approach also maintained greater diversity in the population compared to the normal elitism approach. Figure 4.3-10 shows the average from three runs using pairwise elitism. It demonstrates how the elite member’s fitness is kept almost a constant value greater than that of the average fitness, indicating a reasonable level of diversity throughout the evolution of the population.



26 Figure 4.3-10: Evolution of a path follower using pairwise elitism

Figure 4.3-11 shows the average of three runs using only normal elitism. It can be seen that most of the diversity is lost, in this case around epoch 20. Any improvements in the elite member once this diversity is lost are a result of the mutation operator.



27 Figure 4.3-11: Evolution of a path follower using normal elitism

4.4 Vision

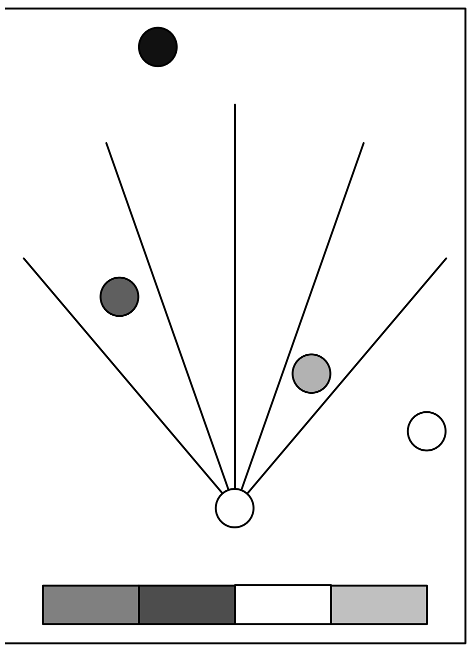
4.4.1 Vision Implementation details

A simple vision system was developed as a means of representing more realistic simulation models to mimic what would actually be used in real world applications. The provision for vision also allows another form of communication if entities have the ability to change the colour they display to other entities.

Though vision was developed it was not part of the major test case, explained in section 5.3, due to the time restrictions of the overall project. Evolutionary learning results were obtained that required vision but not using colour changing as a form of communication.

Vision was implemented by assigning an entity a field of view and a number of segments within the field of view referred to as ‘eyes’. With the direction of the entity known it can be calculated whether other objects in the simulation model are visible and, if so, which eye would ‘view’ the object. A mapping can then be defined from what is seen by the eyes to a number of inputs for a network.
For example with 3 colour components being used to define the possible colour of an object and 4 eyes within the field of view 12 inputs are needed (one for each colour component within each eye). The signal that an eye sees is also scaled relative to the distance to the viewed object to simulate depth cueing.

An example of a possible case of vision is shown in figure 4.4-1



28 Figure 4.4-1: Vision interpretation example

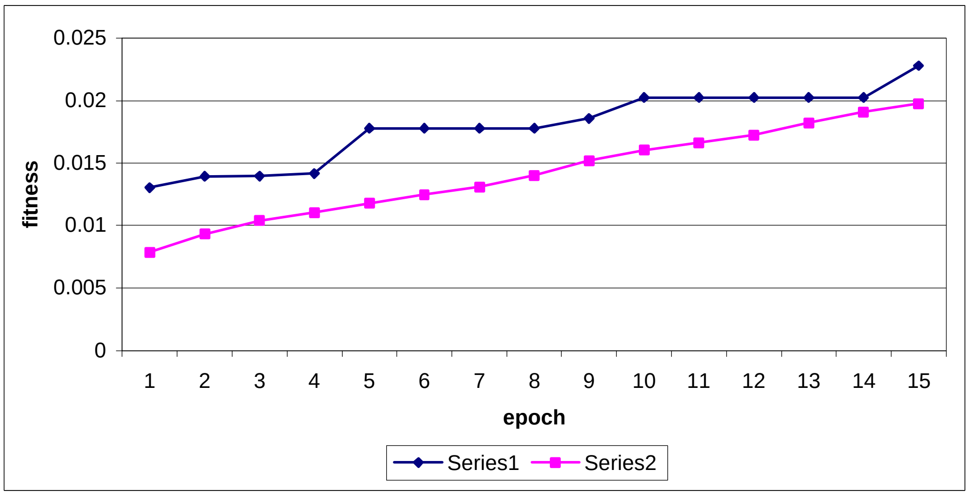
4.4.2 Turning to the red pole

The vision system was tested with the task of turning towards a red object in a room containing other objects of different colours, in this case two green poles, a blue pole and a purple pole (red and blue). The colour of an object was defined in terms of 3 colour components (corresponding to red, green and blue).

The controller for the network used fifteen inputs for the 5 eyes with 3 colour components and 3 outputs for turn left, turn right and don't turn at all. The field of view was 90 degrees with fitness defined as a function of the angle between the direction the entity is facing and the angle to the red pole. This angle was calculated each turn and summed over the entire simulation to give one fitness value. Since it was required that low values of this angle represented a good behaviour the fitness was inverted when the simulation was completed.

To incorporate some degree of non-determinism the entity started a constant distance from the red pole but with a direction defined within +/- 40 degrees of facing it. This ensured that the red pole was within the field of view of the entity at the beginning of the simulation. Since this starting angle changed each entity was tested 4 times with an average performance considered.

Figure 4.4-2 shows the average of three executions of the evolution for learning this task. A single population of 30 members was maintained with the evolution running for a total of 15 epochs and each entity being given 20 turns per trial.



29 Figure 4.4-2: Evolution of an entity using vision to turn towards a target object

Convergence occurred with the elite member able to turn towards the red pole in each simulation. Even though the entity had the ability to not turn in almost all simulations the evolved entity turned to face the red pole and then repeated oscillating between turning left and right. It is hypothesised that this behaviour can be attributed to the fact that the system was trained to in some cases turn left towards the pole and in other cases turn right. Hence the outputs strengths for turning left and right were much stronger then that of not turning. Once the entity had turned left[19] to the red pole then the strength of turning left became less and the turning right signal became dominant. Once the entity had turned right the signal for turning left again became the most dominant and the entity turned back, oscillating between the two.

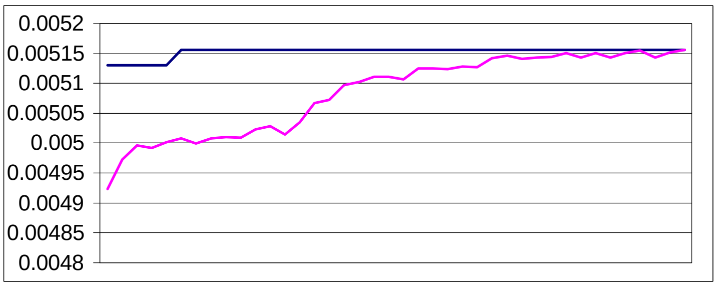
4.4.3 Moving to the red pole

To force the system out of this oscillation process the system was changed. The not-turning decision was replaced by the decision to instead walk forward with the fitness changed to be a function of the distance from the red pole (to be minimised so again the distance summed over all epochs was inverted). This time the system was unable to evolve a performer for the task.

Figure 4.4-3 shows an average of 3 runs for learning this task with a population size of 100 members and an evolution time of 40 generations. Though the graph shows

[19] Without loss of generality.

convergence of the average it can be seen the elite member improves very little with the evolved behaviour simply always walking forward. This is a prime example of how pre-mature convergence means it is possible for sub standard members in the initial population to be able to dominate. In this case the elite members were members with extremely high connections to the walk forward output resulting in the constant behaviour of moving forward. With the entity facing in generally the correct direction walking forward gave a high enough fitness so that the entities trying to learn to turn as well as moving forward were unable to beat those who only walked forward.



30 Figure 4.4-3: Evolution of an entity using vision to walk towards a target object

5 Results with communication

5.1 Hearing facilities

The most intuitive way to implement communication between distinct networks is to reserve a number of nodes in the input and output layers for the purpose of communication. A number of issues arise from the concept of using communication dealing mainly with the nature of the simulation model. For example if one member makes a broadcast, which of the others in the simulation receive it? If two members are to broadcast at the same time how are the messages resolved to the one set of inputs? Also with the simulation occurring with discrete time steps only one entity can be considered to be moving at any time, bringing up questions dealing with *when* other entities hear the broadcast.

Communication was implemented by using a temporary buffer in the world. Since only quite simple models were tested this buffer effectively allowed one entity to hear the broadcast made by the previous entity that had moved. This allows only communication from one entity to one other entity but was not further refined since it was all that was required.

5.2 Migration testing

Migration was implemented as specified before (section 3.5.2) with a number of sub-populations maintained and parameters defined for migration frequency and migration amount. Migration was performed cyclically to remove any favouring of centrally positioned individuals.

As a test of the migration system and how it could improve diversity a simple homogenous communication model was developed. An entity had two copies cloned and placed pseudo randomly[20] in the same virtual environment. Inputs to the controller were its current position and a message of length two from the other clone. Outputs were reserved for the decision to move[21] and for broadcasting a message to the other clone. With the task being for the two clones to move together the only way of achieving a high fitness without explicit knowledge of each others position was for the two to broadcast to each other some function of their own location[22].

The first trials performed could not evolve sensible behaviours due to a small fault in the simulation model. It is interesting to note the error though as another example of greedy optimisation of a fitness function giving an unexpected behaviour. When first trialed the elite behaviour, evolved in less than 10 epochs, was simply for both clones to move always towards the east[23]. With both entities moving the same direction and hence maintaining the same distance apart, it was unsure how this was given a high fitness. The problem lied in the size of the simulated world. It turned out that an entity was able to reach the boundary of the world in the number of moves allocated to it for each simulation regardless of where it randomly started. When an entity reached the edge they were kept there instead of wrapping around. Both then learned to simply move to the right wall and become stuck there, relatively close to each other and hence obtaining a high fitness. Yet another example of sub-standard random individuals at the beginning of an evolution dominated the population early. Increasing the size of the simulation world solved this problem for use in later examples.

Three sub-populations, each consisting of 100 members, were maintained for the next trial. Firstly an execution was performed that included no migration between the three

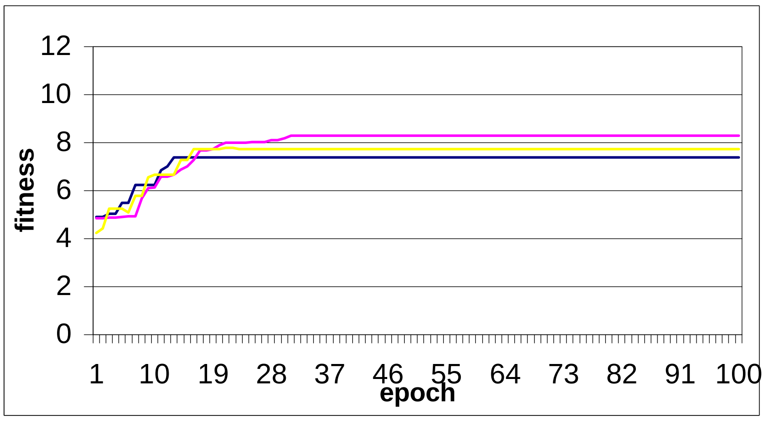
[20] Randomly positioned around each other so that the distance at the start of the simulation was constant. This ensured there that each entity had an equal start while still retaining some degree of non-determinism in the model.

[21] In this case being the 4 directions north, south, east and west

[22] Though it turns out this was not the case!

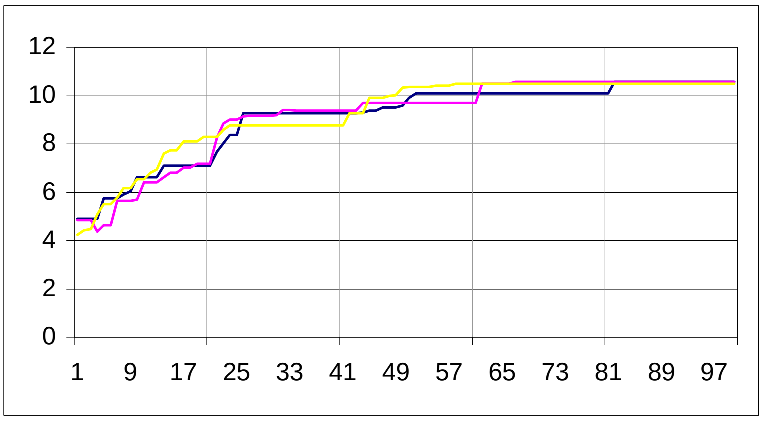
[23] Though without loss of generality the observed behaviour could have been to move any direction.

populations, described by figure 5.2-1 As can be seen all three populations converged at around the same time with a similar fitness for the element member.



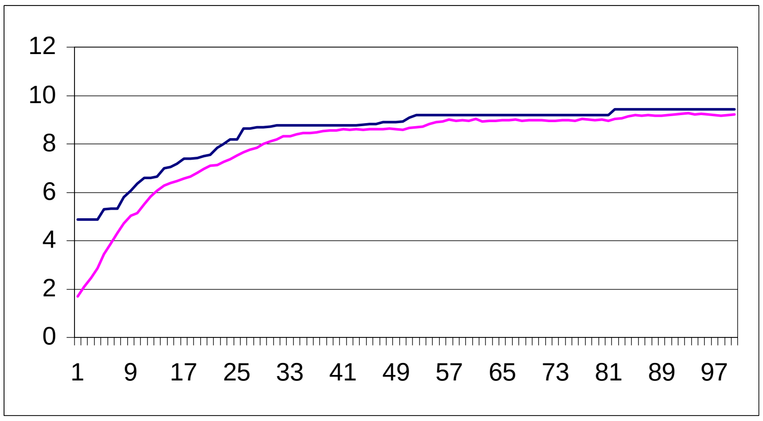
31 Figure 5.2-1: Evolving three sub-populations without migration

The simulation was then run again with a migration this time being applied. The migration rate was chosen to be each 20 epochs since convergence occurred at around this time in the previous test. Each migration moved 5 individuals cyclically choosing all the same numbered members so that none where unfairly lost (as discussed in section 3.5.2). Figure 5.2-2 shows the result from an execution with the vertical lines placed corresponding to the epochs when migration occurred. It can be seen that the migration proved beneficial each time it was applied noting that before each migration instance each sub-population had reached a stable state. It also gave a higher overall converged value with each population having the same elite members fitness (and it turns out the same elite member as expected).



32 Figure 5.2-2: Evolving three sub-populations with migration

Even though having migration gave a better result it was unsure how much was gained from migration explicitly. In the case without migration each member only had it's own sub-population of 100 to breed with. When migration is incorporated the breeding population can be considered to be all 300 members[24]. A further test was then constructed that tested one single population of 300 individuals instead of 100 to see what difference a larger population had on this test problem. Figure 5.2-3 is the result of this evolution and has interesting implications. As expected it performed better than the single populations of 100 members in approximately the same number of epochs but did not outperform the instances of maintaining the separate sub-populations in the same time frame of 100 epochs. For this problem then it shows that maintaining sub-populations can outperform a single population the size of the sub-populations combined.



33 Figure 5.2-3: Evolving all three sub-populations as one single population

Though this example shows how migration can work well it still did not evolve a behaviour in the individuals that was expected. Though the expected results were that the entities would evolve a means of broadcasting to each other some function of their location, the entities actually paid little attention to the message passing. Instead a system was evolved where the entity controller learnt to just move towards one location. Since both entities present are clones then both move to the same location

[24] Though a member only has the chance to breed with the whole population if it is involved with each migration instance.

reducing the distance apart and hence satisfying the requirements of the fitness function. Once again this is example of an unexpected behaviour due to the extremely greedy nature of the evolutionary paradigm along with a loosely defined fitness function.

5.3 Major test case model description

For a major test case a game of "follow the leader" was chosen. This simulation game includes two or more entities, one of which is the 'leader' with the others being 'followers'. The goal of the game is that the followers must all move towards the position of the leader[25]. This model is similar in nature to the previous example but uses two distinct types of individuals that require different behaviours. It was hoped that this distinction would this time force a need for communication as opposed to both just moving to the one position.

All entities were again made aware of their global position in the simulated world and as before had no explicit knowledge of the position of any other entities. Two nodes in the input and output layers were reserved for the purpose of communication between entities. This amount was chosen to correspond to the two coordinates used to determine the position of an entity. One possible solution for using this communication could then be to assign one message position for relaying the details of each coordinate. Inversion was implemented at this time though without specific data gathered on the effect of inversion it is unsure of the effect inversion had.

5.4 The homogenous approach

The homogenous approach uses a single population for evolving controllers that learn the task of being both a leader and a follower. The population is split into a number of sub-populations with a migration system used. Since both the behaviour for a leader and a follower needs to be learnt there needs be provision in the input and output layers of each controller for the requirements of both tasks. Along with this there is required some means of informing the network which role it should play.

[25] Due to time restraints the simulation was only tested using a leader that remained stationary throughout a simulation execution. For this reason perhaps "*go to* the leader" would have been a more accurate name.

The inputs required were then…

- 1 node for specifying whether this entity should act as a leader or follower. This was implemented by hard wiring a signal of 1 to leader controllers and a signal of 0 to follower controllers.
- 2 nodes for representing the position of the entity. With the size of the world having co-ordinates ranging from 0 to 1. Both entities needed to make use of these signals. The leader needed to map them in some way to the outputs reserved for messages and the follower needed to use this along with the incoming message to decide in which direction to move.
- 2 nodes for the actual receiving of messages. Since the follower only used these the leader had them hard wired to zero[26].

The outputs required were…

- 4 for the possible decisions to move 'north', 'east', 'south' and 'west'. Since the leader was not implemented to move these signals were ignored by leader controllers.
- 2 nodes for the broadcasting of messages. This time these nodes are only used by the leader controllers and ignored by the follower controllers.

Testing of the homogenous type of controller was relativity straightforward. Two clones were constructed and placed in a simulated environment with one being assigned to act as the leader and the other designated as the follower.

Fitness was defined as the distance between the leader and follower summed over each turn of a simulation run. Again as this value needed to be minimised the complete sum was inverted.

[26] A possible extension of the problem here could be to treat these as normal input of the message signal so that the leader would have to leader to ignore them.

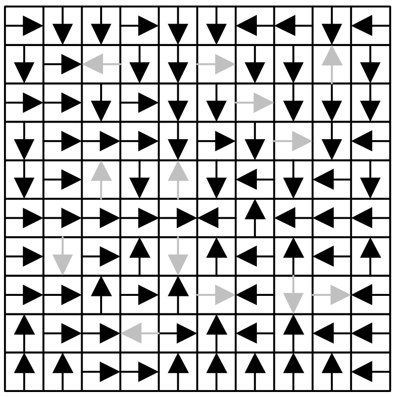
Note that in this case creating multiple instances of the follower in the simulation model can improve the accuracy of testing of the follower role. With multiple followers in different positions a better representation of how the follower works can be obtained. Multiple followers are also easy to incorporate into the fitness function by summing the distance from the leader to each follower.

5.4.1 Homogeneous evolution results

The initial tested simulation used a single population divided into 5 groups of 30 individuals. A crossover rate of 0.8 was defined with a mutation rate of 0.01 and inversion rate of 0.001. Unfortunately a number of tests all showed the system was unable to evolve a system of communication between the leader and the follower even though it displayed normal, albeit slow, improvement and convergence of elite members and average member fitness values. Even still an interesting behaviour was observed where the overall elite member from a number of program executions ignored all communication but managed to evolve followers that moved generally towards the centre of the simulated world. Again this shows a case of the system not being able to evolve the desired result but still being able to define a behaviour that satisfied the fitness function.

It was decided one key factor that caused the system to fail was the difference in complexity of the networks required for a leader and a follower. Where as the leader ideally must just repeat its position, the follower needs to interpret its position as well as the message from the leader and decide on a direction to move. This is made especially difficult for the follower since while it is evolving the leader also is evolving and hence initially gives garbage values as its message to the follower.

Figure 5.4-1 shows the decisions by the elite follower on how to move. This figure was obtained by placing the entity evolved in various positions on a grid and for each position placing the leader in 20 random positions. All arrows show the decision made with black arrows indicating a move towards the centre and the light grey arrows indicating a move away from the centre. A majority of black arrows indicates that the behaviour evolved was to move towards the centre of the grid which is the effective average position of the leader given a number of random placements.



34 Figure 5.4-1: The movement decision of a follower

5.5 The heterogenous approach

In this case the model task was unchanged but separate populations were maintained for the roles of leader and follower. Each population was again broken into sub-populations to maintain diversity with migration used between sub-populations of the same population (ie no breeding between leaders and followers)

Recall that the heterogenous approach has a number of advantages and disadvantages when applied to a team problem.

5.5.1 Population management

The major advantage is that the complexity of a single network is reduced. In the homogeneous case of ‘follow the leader’ it was required that each single member had to learn both the tasks of being a leader and a follower. By using a heterogenous approach we evolve networks that are more specialised for the simple tasks of being only a leader and only a follower. By maintaining separate populations we also gain control over the relative time spent evolving each distinct type of team member. For example in the case of ‘follow the leader’ a large population of followers can be maintained to reflect the need for more time to be spent on them since they require more complex networks.

Unfortunately there are associated disadvantages with using heterogenous evolution. As stated before (section 2.5.1) there are problems with the selection of the members to make up a team and also the allocation of fitness calculated for each attempt at the task.

5.5.2 Team selection

Recall that when using team based problems determining the fitness of an individual requires an entire team. Testing a single member thus requires that other individuals be chosen. The result of the fitness calculation then becomes dependent on the ability of the other team members, not just the ability of the member being tested. This brings up the possibility of a good entity being assigned a bad fitness just because it had poorly performing team members. Two ways to overcome this problem have been tried with varying results defined as follows…

5.5.2.1 Elite team completion

One approach is to complete the team with the relevant elite members from each population. This can be on the scale of an entire population, for example if testing a follower then complete the team by adding the overall elite member from the leader population. In this case since each member is tested only with specific other members, namely the elite members, it was found this technique converged quickly on a sub-optimal solution, mainly due to a lack of diversity of the teams constructed. Alternatively we can add a slight random element by choosing to complete the team with the elite member of a random sub-population of the leader population. This gave more diverse results in the case where there were enough sub-populations but included a completely random event being the selection of which sub-population to use. As each sub-population converged this technique became effectively the same as selecting the overall elite member of the entire population.

5.5.2.2 Non-deterministic random team completion

For a more non-deterministic team we can use the standard selection method applied normally to each sub-population to choose members to complete the team. Since this allows the possibility of every member having a chance to be in each team[27] it should be performed several times for each test with an average taken.

5.5.3 Heterogenous evolution results

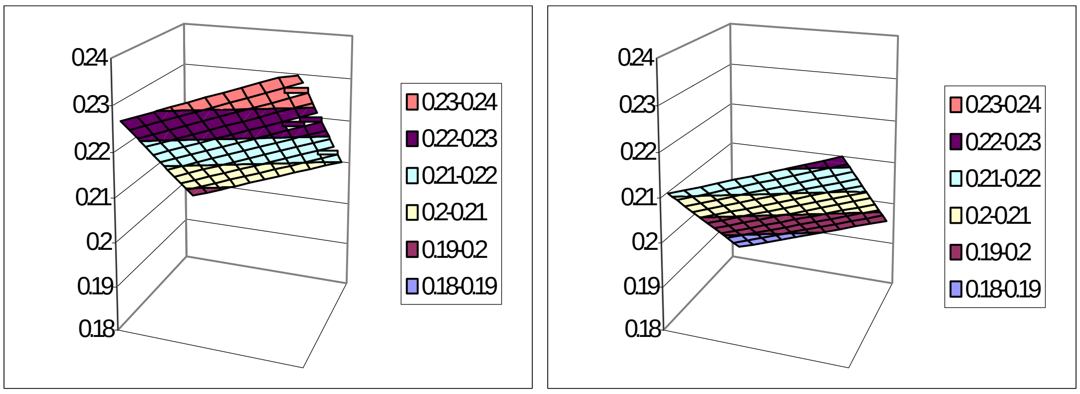
It was found that this approach had trouble using the communication to complete the task.

[27] Proportional to its fitness relative to the other members of its population.

Two types of behaviours were observed as…

1. The followers ignored the output of the leader and moved again to approximately the middle position. When this happened it was the case that one type of follower was dominant early on and the leaders had trouble working with the proper followers in evolving a communication. This is similar to the example of evolution described in section 5.2.
2. The followers and leaders only learnt to express one coordinate in the two decision slots. The followers moved to obtain the same value as the leader in one coordinate but not the other.

An example of the output from a leader is shown in figure 5.5-1. Taking the elite leader evolved and testing the output it gave from various positions on a grid generated these figures. The figures have the x and y axes representing the position of the leader and the z-axis representing the output given. The left shows the values generated on the first output communication node and the right shows the values from the second output communication node.



35 Figure 5.5-1: Communication output of the leader network

It was expected that each output nodes would be allocated to a separate co-ordinate though it can be seen this is not the case.

No behaviours were observed where communication was learnt accurately for both co-ordinates. Extending the number of slots reserved for the message from 2 to 3 slowed the evolution but still gave no examples of good communication. A more accurate fitness function needs to be developed along with a more complex simulation environment specialised for this problem.

These results are strongly co-evolutionary in the sense that combinations of leaders and followers from different executions can not perform together. This is due to the functionally equivalent communications evolved that are incompatible across different executions.

6 Conclusions and further work

It seems that the results gained were strong in the aspects general to all evolutionary techniques (such as migration as a means of maintaining diversity). However the aspects of the project dealing with the evolution of communication did not perform as well as was initially expected.

It was the aim of the project to develop a system as generic as possible, applicable to a wide number of communication applications. It was initially considered important to be able to develop a communication system for a problem without having to input explicit knowledge of how the communication would be performed and reward only in terms of performance. It seems though that desired behaviours were obtained only from problem examples using a very precise fitness function. Such a complex fitness function in many ways outweighs that fact that minimal input is required towards how the communication would act and hence how behaviours should be rewarded.

Any such simulation system relies on complexity being apparent in either the fitness function or the simulation model. Most of the problems studied use an extremely simple simulation model and hence required a complex fitness function so individuals in a population could be correctly and fairly graded against each other. It is believed that moving the complexity of the problem from the fitness function to the simulation model would give results where correct behaviours were evolved from more simple fitness functions. The one case where good results were obtained from a very simple fitness function was in the case of applying vision where the simulation model was quite complex.

The idea of having minimal input into how the communication system will work works though is unrealistic in any real world application. Evolutionary techniques most definitely benefit from having general problem specific knowledge as part of the system (such as the selective positioning genetic operators, useless in any other application not working on the type of networks developed for this project). In hindsight it is believed that better and more complex results could have been obtained by allowing some provision for entering details of the desired communication system to be used. This could be in terms of using some form of back-propagation to ‘steer’ the evolution in the correct direction or perhaps by abstracting to common higher level communication ideas such as ‘turn towards your left’ or ‘move towards me’.

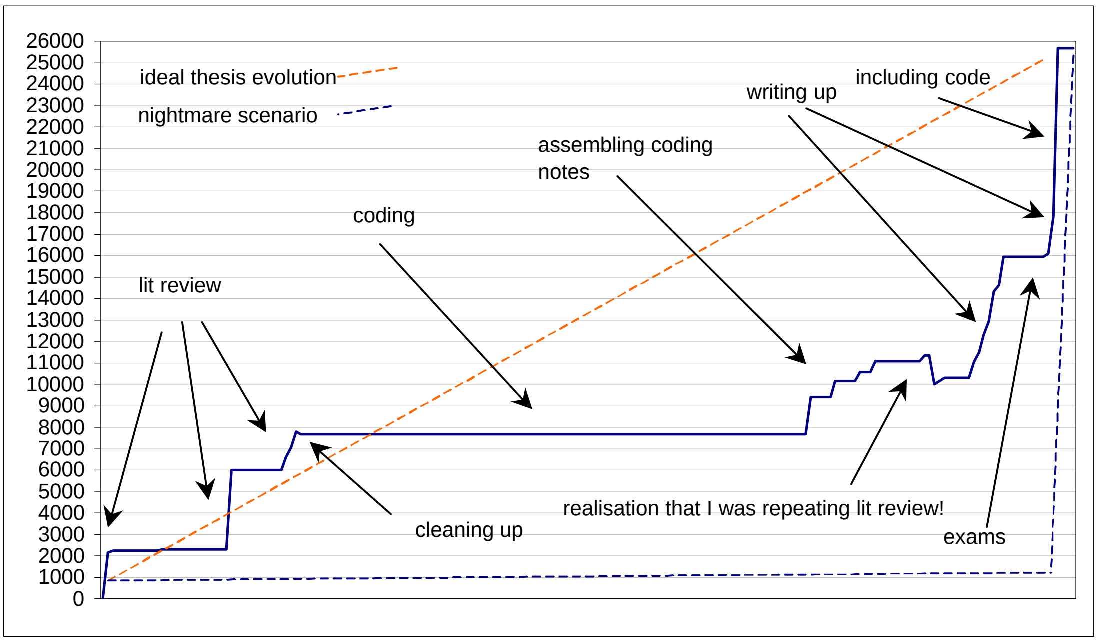
Though neural networks worked a means of providing an object for the evolutionary system to work upon it seems they could have been refined more to work with the system, not just refining the system to work with the networks. Most results gave non-standard network designs where knowledge was stored in terms of usually of one hidden node per problem aspect instead of being distributed across the entire network topology. This is to be expected since crossover implicitly requires that all knowledge is in distinct areas so it can isolated and combined.

In general it is felt that the project had many successes and a number of possible extensions and aspects of possible further work.

1. Firstly the complexity of the simulation model needs to be extended to relieve the pressure for the need for a overly precise fitness function. Since it has already been implemented and tested vision would be a good concept to incorporate into a system that requires communication.
2. Evolutionary techniques are also strong in adapting solutions to keep up with a dynamically changing model. A number of aspects with the communication and overall evolutionary system could be researched in terms of dynamic problems where the communication system could not be static.
3. The neural network implementation needs to be refined so that it is more specially suited to the strong and weak points of the evolutionary system. It could also include further complexity of possible network designs to incorporate multiple hidden layers, non fully connected architectures and recurrent links.
4. More input can be added by incorporating some means of back propagation as a means of directing the evolution. In this way individuals evolved from one execution could perform and make teams with individuals from other evolution executions. This is feasible in simple problems where we can outline the desired communication but in more complex problems we may not be able to define where we need to ‘steer’ the evolution to.

The lighter side of a thesis

On the lighter side of things this graph shows the evolution of a thesis, namely this one! This is a good chance for people to catch me out by checking whether I was actually doing work on whatever night! Thanks for reading this far at least…



Bibliography

Angeline, P., Saunders, G., Pollack, J., 1994 “An evolutionary algorithm that constructs recurrent neural networks” *IEEE Transactions on neural networks*, (5)1 :54-65, January

Belew, R., McInerney, J., Schraudolph, N., 1989, “Evolving networks: Using genetic algorithm with connectionist learning.” Technical report CS TR #95-01, Artificial Intelligence group, Iowa State University, January.

Branke, J., 1995, “Evolutionary algorithms for neural network design and training” *Proceedings of the 1st Nordic workshop on genetic algorithms and its applications.* Vaasa, Finland.

Braun, H., Weisbrod. J., 1993, “Evolving neural feedforward networks” *Proceedings of the conference on artificial neural networks and genetic algorithms.* Springer Verlag, :25-32

Cliff, D., Harvey, I., Husbands, P., 1993, “Incremental evolution of neural network architecture for adaptive behaviour” *Proceedings of the first European Symposium on Artificial Neural Networks.* :39-44

Collier, P., Personal communication.

Collins, R., Jefferson, D., 1991, “AntFarm: Towards Simulated Evolution”, *Artificial Life II*, Addison-Wesley :579-601

Dalton, C., Personal communication.

Dalton, C., “Genetic algorithms against the crozzle puzzle.” An honours thesis submitted 1995

Esparcia-Alcazar, A., Sharman, K., 1995 “Evolving Recurrent Neural Network Architectures by Genetic Programming” Department of Electronics & Electrical Engineering, University of Glasgow. Glasgow, Scotland.

Fraser, A., 1962, “Simulation of genetic systems: Journal of Theoretical Biology” Vol 3. :329-346

Fogel, L., 1962, “Towards Inductive Inference Automata” Technical report GDA-ERR-AN-222, General Dynamics, San Diego.

Fogel, L., 1966, “On the design of Conscious Automata” Final report under contract #AF49(638)-1651, AFOSR, Arlington, VA. USA.

Fogel, L., Burgin, G., 1969, “Competitive Goal-seeking through Evolutionary Programming” Final report under contract #AF19(628)-5927, Air Force Cambridge Research Labs.

Fogel, D., Fogel, L., Porto, V., 1990 “Evolving Neural Networks” *Biological Cybernetics* Springer-Verlag vol 63 :487-493

Fogel, D., 1993, “On the philosophical differences between evolutionary algorithms and genetic algorithms” *Proceedings of the second annual conference on evolutionary programming*

Fogel, D., 1996, *Evolutionary computation: towards a new philosophy of machine intelligence*

Grefenstette. J., 1986 “Optimisation of Control Parameters for Genetic Algorithms” IEEE Trans. Sys., Man., Cybern., Vol6 :122-128

Goldberg, D., 1989, *Genetic Algorithms in Search, Optimisation and Machine Learning* Reading, MA: Addison Wesley.

Gruau, F., 1994, “Genetic micro programming of Neural Networks.” *Advance in Genetic Programming,* The MIT Press.

Grefenstette, J., 1994, “Evolutionary Algorithms in Robotics” *Proceedings of the International Symposium on Robotics and Manufacturing.* August :14-18

Grefenstette, J., Shultz, A., 1994, “An Evolutionary Approach to Learning in Robots.” *Proceedings of the machine learning workshop on robot learning, 11th International Conference on Machine Learning.* :65-72, New Brunswick, New Jersey

Haynes, T., Sen, S., Schoenefeld, D., Wainwright, R., 1995, “Evolving a team” *Working notes of the AAAI-95 Fall Symposium on Genetic Programming.* :23-30. Eric Siegel and John Koza. AAAI press.

Holland, J., 1962, “Outline for a logical theory of adaptive systems” *Journal of the association for computing machinery*. New York NY. USA. Association of computing machinery, Vol 3. :297-314.

Holland, J., 1975, *Adaptation in Natural and Artificial Systems*. Ann Arbor: University of Michigan Press.

Koza, J., 1992, *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* Cambridge, MA: The MIT press

Koza, J., 1994, *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge, MA: The MIT press

Kursawe, F., “Evolution Straegies for Vector Optimization”. *University of Dortmund, Germany.* Email kursawe@LS11.infomatik.uni-dortmund.de

Lewis, M., Fagg, A., Solidum, A., 1992, “Genetic programming approach to the construction of a neural network for control of a walking robot” *IEEE International Conference on Robotics and Automation,* vol 3 May, :2618-2623 IEEE Computing Society Press.

Luger, G., Stubblefield, W., 1993, *Artificial Intelligence: structures and strategies for complex problem solving*. The Benjamin/Cumming publishing company, inc., Redwood City, California. USA.

Luke, S., Spector, L., 1996, “Evolving teamwork and coordination with genetic programming”
The genetic programming 96 conference proceedings, Stanford, July 1996

Maher, M., Poon, J., 1995, “Co-evolution of the Fitness Function and Design Solution for Design Exploration” *IEEE international conference on Neural networks and evolutionary computation* :240-244

Marshall, S., Harrison, R., 1991, “Optimisation and training of feedforward neural networks by genetic algorithms” *Proceedings of the second International Conference on Artificial Neural Networks,* :39-43

Masters, T., 1993, *Practical neural network recipes in C++,* Academic Press, San Diego, USA

Michalewicz, Z., 1992, G*enetic Algorithms + Data Structures = Evolution Programs* Springer Verlag, New York, USA.

Miller, G., Todd, P., Hegde. S., 1989, “Designing neural networks using genetic algorithms.” *Proceedings of the third international conference on genetic algorithms,* Schaffer. J., Arlington, :379-384

Nagahashi, H., Niwa, A., Agui, T., 1995, “Competition and Mutualism in a Simulation of a Adaptive Artifical Organisms” *IEEE international conference on Neural networks and evolutionary computation* :695-700

Rechenberg, I., 1965, “Cybernetic Solution Path of an Experimental Problem.” Royal Aircraft Establishment, Library Translation #1122, August.

Reynolds, C., 1993 “An evolved, vision based behavioral model of coordinated group motion.” *Proceedings of the second International Conference on Simulation of Adaptive Behaviour.* :384-392, Jean-Arcady Meyer, Cambridge, MA. The MIT press.

Saha, S., Christensen, J., 1994, “Genetic design of sparse feedforward neural networks.” *Information Sciences,* (79) :191-200

Salama, R., Hingston, P., 1995, “Evolving neural network controllers” *IEEE international conference on Neural networks and evolutionary computation* :579-583

Schaffer, J., Whitley, D., Eshelman, L., 1992, “Combinations of genetic algorithms and neural netwirks: A survey of the state of the art.” *Proceedings of the International Workshop on Combinations of Genetic Algorithms and Neural Networks.* June :1-37

Schraudolph, N., Belew R., 1992, “Dynamic Parameter Encoding for Genetic Algorithms,” *Machine Learning*, vol 9:1 :9-21

Schwefel, H., 1965, *Numerical Optimisation of Computer Models.* Chichester, John Wiley.

Schultz, A., 1994, "Learning Robot Behaviours using Genetic Algorithms" *Proceedings of the International Symposium on Robotics and Manufacturing.* August :14-18

Schultz, A., Grefenstette, J., 1994, "Evolving Robot Behaviours" *Navy Center for Applied Research in Artificial Intelligence, Naval Research Laboratory.*

Spector, L., 1996, "Simultaneous Evolution of Programs and their Control Strcutures." In *Advances in Genetic Programming 2,* edited by P. Angeline and K. Kinnear. Cambridge, MA: The MIT press.

Thierens, D., Suykens, J., Vandewalle, J., 1993, "Genetic weight optimisation of a feedforward neural network controller." *Proceedings of the Conference on Artifical Neural Networks and Genetic Algorithms.* Springer Verlag, :247-283

Utrecht. U., Trint. K., 1994, "Mutation operators for structure evolution of neural networks" *Parallel Problem solving from Nature, Workshop-proceedings.* Springer Verlag, :492-501

Vamplew, P., Personal communication.

Wasserman, P., 1993, *Advanced Methods in Neural Computing* Van Nostrand Reinhold New York NY. USA.

Whitley, D., Starkweather. T., Bogart, C., 1990, "Genetic algorithms and neural networks: optimising connections and connectivity" *Parallel computing.* 14 :347-361

Wieland, A., 1992, "Evolving neural network controllers for unstable systems" *International Joint Conference on Neural Networks*, vol 2 :667-673. IEEE computing press

Yao, X., 1996, "Evolutionary Artifical Neural Networks" *Department Of Computer Science Australian Defence Force Academy*

Yoon, B., Holmes, D., Langholz G., Kandel, A., 1994, "Efficient genetic algorithms for training layered feedforward neural networks." *Information Sciences.* 76 :67-85

Glossary

Chromosome	DNA collection of genes that is the encoding the evolutionary system uses. Defines exactly a corresponding network.
Clones	Members of a team whose controllers were generated from the single common DNA resulting in identical same deterministic behaviour
Entity	An agent controlled by a network defined by a chromosome that acts in a simulation to define a fitness for the corresponding chromosome.
Epoch	An instance of a single breeding within a population.
Gene	Subsection of a chromosome that defines an individual node in the hidden layer.
Genotype	The representation on an individual in terms of a chromosome structure. This representation includes excessive information stored in hidden node information that is not exhibited
Individual	See entity
Member	A single DNA's association with a specific population or sub-population. Membership can change across sub-populations due to migration but not from population to population. Eg A certain DNA is a member of sub-population 3.
Phenotype	The representation of a chromosome in terms of the neural network it represents. It is the phenotypic representation that is tested for the allocation of fitness values

Appendix A: Program Code

Main program

```
/*      sim manager v5.2
                heterogenous popn model for 'follow the leader'


        Matthew Kelcey
        Honours Research Code.
*/

#include "world3.h"
#include "popn.h"
#include <iostream.h>  //screen output
#include <fstream.h>          //file handling
#include <time.h>             //for determining a 'random' seed

//evolution constants
const int numOfGens=200;              //#generations for evolution
const int outputFreq=1;               //frequency of outputing fitness info to file
//migration specific constants
const int migrationRate=100;  //rate which migration occurs,
                                                            //every migrationRate
epoches.

//////////////////
//global variables
int i,j,s,p;                                  //global loop variables
World *earth;                         //where entities are tested
Popn *leaders,*followers;     //populations (including sub popns)

void main(void) {
        //make the world
        earth = new World();

        ////create the initial populations
        //leader popn has 4 sub-populations each with 50 members
        //migration transfers 5 individuals at a time
        int leaderIn = 1+gps;
        int leaderOut = messLength;
        //the square root of the produce of #input and #output
        //nodes is a good starting number to have. *2 since
        //on average only half will be active
        int leaderHidden = (int)(2*sqrt(leaderIn*leaderOut));
        leaders= new Popn(earth,leaderIn,leaderHidden,leaderOut,4,50,5);
        //read in from file a previously evolved population
        //leaders->readFrom("leaders.pop");

        //follower popn has 4 sub-populations each with 100 members
        //migration transfers 10 individuals at a time
        int follIn = 1+gps+messLength;
        int follOut = decisions; //for NSEW
        int follHidden = (int)(2*sqrt(follIn*follOut));
        followers = new Popn(earth,follIn,follHidden,follOut,4,100,10);
        //read in from file a previously evolved population
        //followers->readFrom(follow.pop");

        //randomize function
        srand(455);

        //prepare log files for leader and followers evolved fitness values
        ofstream lFitValues("loutput.txt",ios::out);
        ofstream fFitValues("foutput.txt",ios::out);

        //do the cycle of time
        for (int time=0; time<numOfGens; time++) {
                int sub; //subpopulation loop variable
                cout << time << " of " << numOfGens << "  "
                        << ((double)time/numOfGens)*100 << "% " << endl;
                //breed leaders
```

```
               for (sub=0; sub<leaders->numSubPopns; sub++) {
                       leaders->breedNextGen(sub);
                       if (time%outputFreq==0)
                               lFitValues     << leaders->averageRawFitness(sub) << ","
                                                      << leaders->highestRawFitness(sub)
<< ",";
               }; //sub
               //breed followers
               for (sub=0; sub<followers->numSubPopns; sub++) {
                       followers->breedNextGen(sub);
                       if (time%outputFreq==0)
                               fFitValues     << followers->averageRawFitness(sub) <<
","
                                                      << followers-
>highestRawFitness(sub) << ",";
               }; //sub

               //if writing to logfile this epoch need to do endl character now
               if (time%outputFreq==0) {
                       lFitValues << endl;
                       fFitValues << endl;
               };

               //migrate if its time to do so
               if (time%migrationRate==0) {
                       cout << "performing migration" << endl;
                       leaders->migrate();
                       followers->migrate();
               };
       };

       //if required save these populations to file for recalling later
       //leaders->writeTo("leaders.pop");
       //followers->writeTo("followers.pop");

       // kill populations
       delete leaders;
       delete followers;
};
```

World header file

```
/*      WORLD3.H
        world class for simulating in for entity fitness evaluation
        specific for leader and follower problem

        Matthew Kelcey
        Honours Research Code
*/

#ifndef WORLD3_H
#define WORLD3_H

#include "pos.h"
#include "virtudna.h"
#include "entity.h"

//world constants
const int numSimRuns = 5;     //#times dna tested per trial (used when there is
                                                            //some undeterministic
element)
const int simLength = 10;     //life time of the simulation
const int sizeX = 1;                  //keep between 0 and 1 to make inputs to networks
const int sizeY = 1;                          //easy to handle

class World {
public:
        World();
        ~World();

        void display(void);           //display all info about all entities

        void addEntity(eEntityType, vDNA*);

        //run the simulation and return an obtained fitness value
        double runSimulation(int,     //display moves flag
                                 int,      //display entity info flag
                                 int);     //write to file flag

private:
        //vars
        Entity *pLeader,*pFollower;
        double *worldNoises;                  //array to hold world noise information
                                                                    //dynamic since may
have no elements
        double calculatedFitness;
};

#endif
```

World class definition

```
/*      WORLD3.CPP
        adapted from simworld.cpp

        Matthew Kelcey
        Honours Research Code
*/

#include "world3.h"
#include "entity.h"
#include <iostream.h> //for debugging
#include <fstream.h>
#include <string.h>
#include <math.h>
#include <assert.h>

//con and decon
World::World(void) {
        //make and then clear noises array out
        worldNoises = new double[messLength];
        for (int i=0; i<messLength; i++)
                worldNoises[i]=(double)0;

        //allocate space for leader and follower
        pLeader = new Entity();
        pFollower = new Entity();

        //open leaderfile for the first time to flush it
        //prepare file for displaying output of leader
        ofstream leaderOutput;
        leaderOutput.open("leader.txt",ios::out);
        ofstream followerOutput;
        followerOutput.open("follower.txt",ios::out);
}; //world


World::~World(void) {
        //free some memory
        delete [] worldNoises;
        delete pLeader;
        delete pFollower;
}; //~world

void World::display(void) {
        //invoke display on leader and follower
        pLeader->display();
        pFollower->display();

        //wait for user
        cout << "hit an int ";
        int reply; cin >> reply;
}; //display


double doubleAbs(double x) {
        if (x>=0) return x;
                else return -x;
};

void World::addEntity(eEntityType type, vDNA *pDna) {
        if (type==leader)
                pLeader->constructFromDNA(pDna);
        else //type==follower
                pFollower->constructFromDNA(pDna);
};

double World::runSimulation(int fDispMoves,
                                                int fDispEntInfo,
                                                int fWriteToFile) {
        /*
        simulation description
                first animate is the leader,
                        knows position, hears nothing, doesn't move
                        does broacast though (what should map to its position)
                all other animates
                        knows position, hears leaders output
                        moves (hopefully towards leader!), doesn't broadcast
        */

        //reset the calculated fitness for this simulation run
        calculatedFitness=0;

        //wish to repeat whole process a number of times to obtain
        //a fairer representation.

        for (int repeat=0; repeat<numSimRuns; repeat++) {
                //position the entities randomly, but equally spaced
                Position lPos, fPos;   //l=leader, f=follower
                bool validPositions=false;
                while (!validPositions) {
                        //choose leader position as random in sqaure
                        lPos.set(randDouble(1),randDouble(1));
                        //choose follower position as offset from leader by distance 0.5
                        //with stepsize of 0.05 should travel 0.5 in 10 steps
                        double angle=randDouble(twoPi);
                        fPos.set(lPos.x+0.5*cos(angle),lPos.y+0.5*sin(angle));
                        //check if the position of the follower is valid
                        if (fPos.x>0 && fPos.x<1 && fPos.y>0 && fPos.y<1)
                                validPositions=true;
                }; //while !validPositions
                pLeader  ->relocate(lPos,0);
                pFollower->relocate(fPos,0);

                //run the simulation once with these positions
                for     (int time=0; time<simLength; time++) {
                        //move the entities and check for wall collision
                        pLeader->move(leader,worldNoises);
                        if (pLeader->loc.x<0)         pLeader->loc.x=0;
                        if (pLeader->loc.y<0)         pLeader->loc.y=0;
                        if (pLeader->loc.x>maxX)      pLeader->loc.x=maxX;
                        if (pLeader->loc.y>maxY)      pLeader->loc.y=maxY;
                        pFollower->move(follower,worldNoises);
                        if (pFollower->loc.x<0)               pFollower->loc.x=0;
                        if (pFollower->loc.y<0)               pFollower->loc.y=0;
                        if (pFollower->loc.x>maxX)    pFollower->loc.x=maxX;
                        if (pFollower->loc.y>maxY)    pFollower->loc.y=maxY;
                        //update fitness value
                        calculatedFitness += pLeader->loc.distTo(pFollower->loc);
                }; //time loop
        }; //repeat loop

        return (double)calculatedFitness/numSimRuns;
}; //runSimulation
```

Population header file

```
/*      Popn.h: population class
        includes
                preforming of evolutionary steps
                        selection
                        crossover, mutation and inversion of dna strings
                        fitness scaling

        Matthew Kelcey
        Honours Research Code.
*/

#ifndef POPN_H
#define POPN_H

#include "virtuDNA.h"
#include "world3.h"

//simulation prob chances             (values 0-1, 0-never, 0.5 50%, 2-always)
const double introRandom=0.3; //add random member to popn each gen, to replace worst
const double pairwiseElitism=2;       //each next gen must beat the value in the slot
const double elitism=2;                       //highest entity is saved in each
generation

//fitness calculation things
const double scaler=2;                //after rescaling fitness,
                                                              //maxFitness =
scaler*averageFitness

class Popn  {
public:
        //constructor and destructor
        Popn(World*,           //need to know where the population is
                 int,int,int,  //ins, maxhiddens and outs of popn members
                 int,int,int); //numSubPopns, subPopnSize, migrationNumber;
        ~Popn();

        //interface functions
        void display(int);                    //invoke display on all vDNA members,
                                                              //0=all info,
1=fitness only
        vDNA* fetch(int,int);         //return with a ptr to the ith member of
                                                              //the jth subpopn
        vDNA* fetchElite(int);        //return the elite of a subpopn
        void breedNextGen(int);               //perform breeding on sth subpopulation
        void migrate(void);                   //migrate individuals cyclically
        vDNA* select(void);                   //select a member from the whole
population
        void testAllMembers(int);     //test all members of a subpopn
        void calcRawFitness (vDNA*);//calculate the fitness of a popn member

        void dispElite(int); //display info on elite member of given subpopn

        double averageRawFitness(int); //int is which sub popn
        double highestRawFitness(int); //int is which sub popn

        //streaming functions (return success or otherwise)
        int writeTo(char*);           //write the popn to a file
        int readFrom(char*);          //read the popn from a file

        //variables that once were constants
        int subPopnSize;
        int numSubPopns;
        int migrationNumber;

private:
        //popn variables, most dynamically defined once population
        //sizes known gathered constuctor.
        World *pHomePlanet;                   //where this populatoin is (needed for
testing)
        double **selectArray;         //selection arrays for each subpopn
        double *globalSelectArray;    //the gloabl selection array
```

```
	int ins,maxHiddens,outs,dnaLength; //dna variables for members in this
population
	int *eliteMember;	//array of sub-population elite members
	vDNA ***pDna;		//actual members in the population
	vDNA **pTempPopn;	//need a temp array for holding nextgen members
				  	//and for usage in migration

    int fNeedGlobalRecalc;	//flag to indicate that a subpopn has changed its
							//selection array and so
global array must be updated


	//private functions needed to be called only by member functions.
	vDNA* select(int);					//select a member from
subpopulation s
	void calcFitnessStats(Fitness,int);	//determine elite member for subpopn
	void prepareSelectionArray(int);	//needed for selection of members for a
subpopn
};

#endif
```

Population class definition

```
/*      popn.cpp: implementation of the popn class.

        Matthew Kelcey
        Honours Research Code
*/

#include "popn.h"
#include <iostream.h>
#include <fstream.h>
#include <assert.h>

void Popn::dispElite(int i) {
        pDna[i][eliteMember[i]]->display(0);
};

/*procedure for activation of a flag
        chance=0    => always returns 0
        chance=0.5  => returns 1 50% of the time
        chance=2        => always returns 1 */
inline int active(double chance) {
        return (randDouble(1)<chance);
};

//return the average raw fitness of a specified sub population
double Popn::averageRawFitness(int ws) {
        double total=0;
        for (int p=0; p<subPopnSize; p++)
                total+=pDna[ws][p]->fitness[raw];
        return total/subPopnSize;
};

//return the highest raw fitness of a specified sub population
double Popn::highestRawFitness(int ws) {
        int highest=0;
        double highestRF=pDna[ws][0]->fitness[raw];
        for (int p=1; p<subPopnSize; p++)
                if (pDna[ws][p]->fitness[raw] > highestRF) {
                        highest=p;
                        highestRF=pDna[ws][p]->fitness[raw];
                };
        return highestRF;
};

//display the details of the DNA strings in the population
void Popn::display(int disp) { //0=all info , 1=fitness values only
        int s,p;
        for (s=0; s<numSubPopns; s++) {
                cout << "***" << s << "th subpopn" << endl;
                for (p=0; p<subPopnSize; p++) {
                        cout << "member " << p << " of subpop " << s << endl;
                        pDna[s][p]->display(disp);
                };
        };

        //heres the selection arrays
        cout << "and the selection arrays are " << endl;
        for (s=0; s<numSubPopns; s++) {
                for (p=0; p<subPopnSize; p++)
                        cout << selectArray[s][p] << ",";
                cout << endl;
        };
};

//population constructor
Popn::Popn(World *pWhichWorld, int ins_, int maxHiddens_, int outs_,
                              int numSub, int subSize, int migrationNumber) {
        int s,p; //loop variables

        //keep relevant values
        ins = ins_;
        maxHiddens = maxHiddens_;
        outs = outs_;
        numSubPopns = numSub;
        subPopnSize = subSize;
        pHomePlanet = pWhichWorld;
        //dna length used so often store it also
        dnaLength = (1+ins+outs)*maxHiddens;

        //construct the dynamic arrays needed for population management
        //allocate space for elite member array
        eliteMember = new int[numSubPopns];
        //allocate space for select array
        selectArray = new double*[numSubPopns];
        for (s=0; s<numSubPopns; s++)
                selectArray[s] = new double[subPopnSize];
        //allocate space for global selection array
        globalSelectArray = new double[subPopnSize*numSubPopns];
        //allocate space for population dna pointers
        pDna = new vDNA**[numSubPopns];
        for (s=0; s<numSubPopns; s++)
                pDna[s] = new vDNA*[subPopnSize];
        //allocate space for temporary sub population
        pTempPopn = new vDNA*[subPopnSize];

        //create the individual members
        for (p=0; p<subPopnSize; p++) {
                pTempPopn[p]   = new vDNA(ins,maxHiddens,outs);
                for (s=0; s<numSubPopns; s++)
                        pDna[s][p]     = new vDNA(ins,maxHiddens,outs);
        };

        //test members of each subpopn to obtain initial fitness values
        for (s=0; s<numSubPopns; s++)
                testAllMembers(s);
}; //con

//destructor, need to free up heaps of dynamically defined memory
Popn::~Popn()  {
        int p,s; //loop variables

        //kill all the members
        for (p=0; p<subPopnSize; p++) {
                delete pTempPopn[p];
                for (s=0; s<numSubPopns; s++)
                        delete pDna[s][p];
        };

        //remove all space reserved for dynamic arrays
        delete [] eliteMember;
        delete [] globalSelectArray;
        delete [] pTempPopn;
        for (s=0; s<numSubPopns; s++) {
                delete [] selectArray[s];
                delete [] pDna[s];
        }; //s
}; //destr


//select a member from subpopulation s
//if offset is nonzero then use this value instead of a random value
//more intelligent selection rountine. O(n) for whole population selection
vDNA* Popn::select(int whichSpecies) { //s
        //choose the random position
        double pos=randDouble(selectArray[whichSpecies][subPopnSize-1]);

        //check for boundary cases,
        //also ensures s[0] < p < s[n-2]
        if (pos<=selectArray[whichSpecies][0])
                return pDna[whichSpecies][0];                       //the first member
        if (pos>=selectArray[whichSpecies][subPopnSize-2])
                return pDna[whichSpecies][subPopnSize-1];    //the last member

        //make the inital guess at position, = position/average
        int guess=(int)(pos/(selectArray[whichSpecies][subPopnSize-1]

        /subPopnSize));

        //move until its found
        while (1) {
                //check if the guess is now correct
                if ((pos>selectArray[whichSpecies][guess-1])
                        && (pos<=selectArray[whichSpecies][guess]))
                        return pDna[whichSpecies][guess]; //found it!
                //otherwise try guesing one way or the other
                if (pos>selectArray[whichSpecies][guess])
                        guess++; //move to right one place
                else
                        guess--; //move to left one place
        }; //while
};

/*/naive selection rountine, O(n*n) for whole population selection
int oldRouletteSelect(double total, int whichSpecies) {
        int which=0;
        double where=randDouble(total);
        while ((where>=pDna[which][whichSpecies]->fitness[scaled])
                                && (which<subPopnSize-1))
                where-=pDna[which++][whichSpecies]->fitness[scaled];
        return which;
};      //nothing
*/

//select used outside the class by others
//returns a random member from the entire population
vDNA* Popn::select(void) {
        ////simple naive approach
        return select(randInt(numSubPopns));
};

//migrate starting at a random point.
//(do modulo subPopnSize) to treat cyclically.
void Popn::migrate(void) {
        int i,j;        //loop variables
        int lower;      //lower bound of migration range.

        //choose lower bound;
        lower = randInt(subPopnSize);

        //remember the zeroth popn in a temp array
        for (j=lower; j<lower+migrationNumber; j++)
                pDna[0][j%subPopnSize]->copyInto(pTempPopn[j%subPopnSize]);

        //create temporary popn for moving groups
        //starting from 2nd popn (popn group 1)
        for (i=1; i<numSubPopns; i++)
                //move the ith popn into the i-1th population
                for (j=lower; j<lower+migrationNumber; j++)
                        pDna[i][j%subPopnSize]->copyInto(pDna[i-1][j%subPopnSize]);

        //move the zeroth group into the 'numSpecies'th popn
        for (j=lower; j<lower+migrationNumber; j++)
                pTempPopn[j%subPopnSize]->
                                copyInto(pDna[numSubPopns-1][j%subPopnSize]);

        //since members have been moved need to reevaluate
        //the selection array after rescaling the fitness
        //values for each subpopn
        for (int s=0; s<numSubPopns; s++)
                prepareSelectionArray(s);
};

void Popn::calcRawFitness(vDNA *testee, eEntityType leaderOrFollower) {

        //testee->fitness[raw]=(float)0;

        // RUN SIMULATION
        if (testee->numHiddens()==0) //invalid network
                testee->fitness[raw]=0;
        else { //valid network with one or more hidden nodes
                //clear current fitness value
                testee -> fitness[raw] = (float)0;
                //add the testee as a leader or follower
                pHomePlanet->addEntity(leaderOrFollower,testee)
                //run the simulation a number of times,
                //each time with a new patner
                for (sim=0; sim<numSimRuns; sim++) {
```

```
                if (leaderOrFollower==leader)
                        pHomePlanet->addEntity(follower, some other member);
                else
                        pHomePlanet->addEntity(leader, some other member);
                //award all fitness to the testee
                testee -> fitness[raw] += pHomePlanet->runSimulation(0,0,0);
        }; //for sim
        //average fitness value
        testee -> fitness[raw] /= (double)numSimRuns;
}; //else
//*/

/*/TRIVIAL TEST OF ONE INPUT
//MUST HAVE 4inputs AND 3outputs
Network *brain = new Network(testee);
double inputs[4] = {1,0.2,0.3,-0.3};
double trueValues[3] = {0.5,-0.3,0.4};
testee->fitness[raw] = brain->errorMagnitude(inputs,trueValues);
delete brain;
//*/

/*/TEST BY ABILITY TO COPY INPUT STRING ON OUTPUT
testee->fitness[raw]=(float)0;
//make controller
Network *brain = new Network(testee);
//create arrays for testing
double *inputs = new double[testee->ins];
double *trueValues = new double[testee->outs];
inputs[0]=1; //for bias terms
//test simLength times
for (int t=0; t<simLength; t++) {
        //create some random inputs
        for (int i=1; i<ins; i++) {
                inputs[i]=negPos();
                trueValues[i-1]=inputs[i];
        };
        //find errorMagnitude on these inputs
        testee->fitness[raw]+=brain->errorMagnitude(inputs,trueValues);
};
//invert fitness values (we want low errors to mean high fitness)
testee->fitness[raw]=1/testee->fitness[raw];
//free memory
delete brain;
delete [] inputs;
delete [] trueValues;
//*/

/*/TEST BY ALLOCATING FITNESS AS SUM OF DNA STRING VALUES
testee->fitness[raw]=(float)0;
for (int i=0; i<dnaLength; i++)
        testee->fitness[raw]+=testee->piece[i];
//*/

/*/TEST ON TIME SERIES ERROR MAGNITUDE
//reset old fitness value
testee->fitness[raw]=(float)0;
//make controller from dna
Network *controller = new Network(testee);
//test on simple time series
for (int i=0; i<seriesLength; i++)
        controller->addToRaw(controller
                ->errorMagnitude(seriesInputs[i],seriesTrueValues[i]));
//we require small fitness values indicate a fit individual
testee->fitness[raw]=1/testee->fitness[raw];
cout << "fitness is " << testee->fitness[raw] << endl;
//free some memory
delete controller;
//*/
};

//first thing to do when all have been created
void Popn::testAllMembers(int ws) {
        cout << "testing all members of subpopn " << ws << endl;
        for (int s=0; s<numSubPopns; s++)
                for (int p=0; p<subPopnSize; p++)
                        calcRawFitness(pDna[s][p]);
//prep selection array (including conversion from
//raw->scaled fitness values
prepareSelectionArray(ws);
};


//need to ensure raw fitness values have been rescaled to scaled values
void Popn::prepareSelectionArray(int ws) {
int p;
////first need to convert raw values to scaled values

//need to check all values are non-negative
//if some are not rescale on the most negative value while
//retaining the correct level of proportionality
double mostNegative=0; //set to zero value to check against
for (p=0; p<subPopnSize; p++) {
        if (pDna[ws][p]->fitness[raw]<mostNegative)
                mostNegative=pDna[ws][p]->fitness[raw];
        //cout << "mn" << mostNegative << " ";
}; //p
if (mostNegative!=0) //ie value was set previously
        for (p=0; p<subPopnSize; p++)
                pDna[ws][p]->fitness[raw] -= mostNegative;

//find the average and highest value
double highestF = pDna[0][ws]->fitness[raw];
double totalF   = highestF; //ie just first value
double averageF;
eliteMember[ws]=0;

//go through rest of the popn and get actual
//average and highest raw values
for (p=1; p<subPopnSize; p++) {
        totalF  += pDna[ws][p]->fitness[raw];
        //check if this ones the elite member
        if (pDna[ws][p]->fitness[raw] > highestF) {
                highestF = pDna[ws][p]->fitness[raw];
                eliteMember[ws]=p;
        };
};
//obtain average from total
averageF = (double)totalF/subPopnSize;

//calculate constants
double slope=(scaler-1)*averageF/(highestF-averageF);
double constant=averageF*(highestF-scaler*averageF)
                                                                     /(highestF-
averageF);
//need to keep tabs on negative values
//if any exist then shift all so all non-negative
double lowestNegValue=0;
//rescale all other values using constants
//checking for negative values
for (p=0; p<subPopnSize; p++) {
                pDna[ws][p]->fitness[scaled] =
                                slope*pDna[ws][p]->fitness[raw]+constant;
                //make sure none are negative due to scaling, otherwise
                //roulette will fail
                if (pDna[ws][p]->fitness[scaled]<0)
                        if (pDna[ws][p]->fitness[scaled]<lowestNegValue)
                                lowestNegValue=pDna[ws][p]->fitness[scaled];
}; //for

//if there are negatives then shift all values
//to make the most negative zero
if (lowestNegValue!=0) //ie it has changed, by being set above
        for (p=0; p<subPopnSize; p++)
                pDna[ws][p]->fitness[scaled] -= lowestNegValue;
//note: even after rescaling the elite member will stay the same

//////then create actual selection array
//set first member to be first fitness value
selectArray[ws][0] = pDna[ws][0]->fitness[scaled];
//allocate following ones as the sums
for (int i=1; i<subPopnSize; i++)
        selectArray[ws][i] = selectArray[ws][i-1] +
                                                    pDna[ws][i]-
>fitness[scaled];

//since this selection array has changed will
//need to recalc the global selection array
fNeedGlobalRecalc=1;
};


vDNA* Popn::fetch(int whichSpecies, int whichMember) {
return pDna[whichSpecies][whichMember];
};
vDNA* Popn::fetchElite(int whichSpecies) {
return pDna[whichSpecies][eliteMember[whichSpecies]];
};


//perform breeding on sth subpopulation
void Popn::breedNextGen(int ws) {
vDNA *parent;
int p;  //loop iter
int startFrom=0; //will start here if elite selection happens

//do elite member automatic inclusion
if (active(elitism)) {
        pDna[ws][eliteMember[ws]]->copyInto(pTempPopn[0]);
        startFrom=1; //don't want to copy over the new elite member
};

//do the actual breeding of each member
for (p=startFrom; p<subPopnSize; p++) {
        //select a parent member
        parent = select(ws);
        //copy it to the next gen
        parent->copyInto(pTempPopn[p]);

        //test for crossover & mutation
        int fChanged=false; //need to keep since
                                            //may need to reevaluate fitness
        if (active(crossOverRate)) {
                //crossover of parent and another selected parent
                pTempPopn[p]->crossOver(parent,select(ws),2);
                fChanged=true;
        };
        if (active(mutationRate)) {
                pTempPopn[p]->mutate();
                fChanged=true;
        };
        if (active(inversionRate)) {
                pTempPopn[p]->inversion();
                fChanged=true;
        };

        //if its changed then reevaluate its fitness
        if (fChanged)
                calcRawFitness(pTempPopn[p]);

        //do pairwise elitism tests
        if (fChanged &&
                active(pairwiseElitism) &&
                (pTempPopn[p]->fitness[raw] < parent->fitness[raw]))
                        //replace old parent in this slot
                        parent->copyInto(pTempPopn[p]);
};

//perform new random member inclusion to overwrite worse member
if (active(introRandom)) {
        //find the worst member
        int worst              = 0; //for now
        double worstFitness = pTempPopn[0]->fitness[raw];
        for (p=1; p<subPopnSize; p++)
                if (pTempPopn[p]->fitness[raw] < worstFitness) {
                        worst = p;
                        worstFitness = pTempPopn[p]->fitness[raw];
                };//if
        //replace it with a new random dna
        pTempPopn[worst]->randomizeValues();
        //evaluate this new members fitness
        calcRawFitness(pTempPopn[worst]);
};

//copy next gen back to breeding pool
for (p=0; p<subPopnSize; p++)
        pTempPopn[p]->copyInto(pDna[ws][p]);

//rescale fitness values and construct new selection array
prepareSelectionArray(ws);

//so at end of breeding epoch the selection array
//should be up to date (for use by other populations
//when asking to choose a new member)
};

//streaming
int Popn::writeTo(char *fileName) {
//open the file for writing
ofstream output(fileName,ios::out);
if (!output) {
        cout << "error opening " << fileName << " for writing" << endl;
        return 0;
};
//write some general popn info (used for checking)
output  << numSubPopns << " " << subPopnSize
                << " " << dnaLength << " ";
//write all the members data out
for (int s=0; s<numSubPopns; s++)
        for (int p=0; p<subPopnSize; p++) {
                for (int l=0; l<dnaLength; l++)
                        output << pDna[s][p]->piece[l]<< " ";
                output << pDna[s][p]->fitness[0] << " ";
                output << pDna[s][p]->fitness[1] << " ";
        };
output << endl;
//if this far then success
return 1;
};

int Popn::readFrom(char *fileName) {
//open the file for reading
ifstream input(fileName,ios::in);
if (!input) {
        cout << "error opening " << fileName << " for reading" << endl;
        return 0;
};
//read in general popn info
int subpop,popsize,dnalen;
input >> subpop >> popsize >> dnalen;
//compare against runtime constants
if ((subpop!=numSubPopns)||(popsize!=subPopnSize)
                        ||(dnaLength!=dnalen)) {
        cout << "error in file, differing constants" << endl;
        cout << "numSubPopns :" << subpop
                        << " should be " << numSubPopns << endl;
        cout << "subPopnSize :" << popsize
                        << " should be " << subPopnSize << endl;
        cout << "dna length  :" << dnalen
                        << " should be " << dnaLength   << endl;
        return 0;
};

//read all the info in
for (int s=0; s<numSubPopns; s++)
        for (int p=0; p<subPopnSize; p++) {
                for (int l=0; l<dnaLength; l++)
                        input >> pDna[s][p]->piece[l];
                input >> pDna[s][p]->fitness[0];
                input >> pDna[s][p]->fitness[1];
        };

//recreate the selection arrays
for (s=0; s<numSubPopns; s++)
        prepareSelectionArray(s);

//if this far then success
return 1;
};
```

Virtual DNA header file

```
/*	VIRTUDNA.H
	by Matthew Kelcey
	Honours Research Code.
*/

#ifndef VIRTUDNA_H
#define VIRTUDNA_H

#include <stdlib.h> //for random
#include <math.h>	//for sqrt

//dna length flags
const int numEyes = 0;		// number of eyes, 0=no eyes
const int numColourComps = 3; // number of colour componentes, RGB
const int gps = 2;				// 2=xy coords, 0=no gps
const int messLength = 2;		// length of output consider to be the
						// message, 0=>no hearing
const int decisions = 4;		// 4=>nsew, 3=>lrs

////breeding constants
//chance of mutation
const float mutationRate = (float)0.01;
//relative odds of swapping node active
const int mutateActive = 1;
//relative odds of changing a weight (includes bias terms)
const int mutateWeight = 4;
//this is the prob of a new value opposed to
//gaussian changing given a mutation is occuring
const float newValueOrGauss = (float)0.2;

//chance of crossover
const float crossOverRate = (float)0.8;
//rel. odds of crossover point on active position
const int xOverNodes = 3;
//rel. odds of crossover point on first outgoing weight
const int xOverInsOuts = 2;
//rel. oods of crossover point anywhere (could be one of the above though)
const int xOverAnywhere = 1;

//chance of inversion
//acts only on genes
const float inversionRate = (float)0.001;

//to return a random float between -1 & 1
inline double negPos(void) {
	return ((double)rand())/RAND_MAX*2-1;
};

//return a random number between 0 and max (as a double)
inline double randDouble(double max) {
	return (double)(rand()*max/RAND_MAX);
};

//return an int from 0->max-1
inline int randInt(int max) {
	return rand()%max;
};

enum Fitness {raw, scaled};

class vDNA {
public:
	//return how many hidden nodes this dna represents
	int numHiddens(void);
	//set all pieces to random values
	void randomizeValues(void);
	//defaultconstructor, dna sizes defined by constants
	vDNA(int,int,int); //ins, maxHiddens, outs
	//default decon.
	~vDNA();
	//copy con
	vDNA(vDNA&);
```

```
        //debugging displayer ints are inputs and outputs (for formatting)
        void display(int); //0=all, 1=just fitness values
        //copier, why doesn't copycon work?
        void copyInto(vDNA*);
        //mutate the dna
        int mutate(void);
        //crossover things
        int newCrossOverPoint();      //give a new crossover point
        int crossOver(vDNA*,   //other parent
                                vDNA*,       //child
                                int);        //number of crossover points
        //inversion functions
        void swap(int,int); //swap two genes in strand
        void inversion(void);

        //variables
        double *piece;         //actual weights array
        double fitness[2];     //two fitness values, raw and scaled
        int mutateTotal;//= mutateActive+(geneLength-1)*mutateWeight;
        int xTotal;// = xOverNodes+xOverInsOuts+geneLength*xOverAnywhere;
        int ins,maxHiddens,outs;
        int geneLength,length;
};

#endif
```

Virtual DNA class definition

```
/*      VIRTUDNA.CPP
        by Matthew Kelcey
        Honours Research Code.
*/

#include "virtudna.h"
#include <iostream.h> //for debugging
#include "math.h"

//need this since the 'log' in gaussian dies when it gets a zero
inline double spRand(void) {
        double temp=randDouble((double)1);
        if (temp!=0)
                return temp;
        else
                return 0.000001;
};
inline double newGaussian(double mean, float stdDev) {
        return sqrt(-2.0 * log(spRand()))
                        *cos(randDouble((double)6.2831853072))
                        *stdDev+mean;
};

//return the number of hidden nodes this dna would have active
int vDNA::numHiddens(void) {
        int numHiddens=0;
        for(int i=0; i<length; i+=geneLength)
                //check active positions
                numHiddens += (piece[i]>0);
        return numHiddens;
};

//nuke to randomise all values
void vDNA::randomizeValues(void) {
        //put random values in dna
        for (int i=0; i<length; i++)
                piece[i] = (float)(negPos()/2);
        //zero fitness values
        fitness[0] = fitness[1] = (float)0;
};

//constructor
vDNA::vDNA(int ins_, int maxHiddens_, int outs_) {
        //remember constants
        ins=ins_;
        maxHiddens=maxHiddens_;
        outs=outs_;
        //work out a few values accessed often to optimise time
        geneLength = ins+outs+1;
        length = geneLength*maxHiddens;
        mutateTotal = mutateActive+(geneLength-1)*mutateWeight;
        xTotal = xOverNodes+xOverInsOuts+geneLength*xOverAnywhere;
        //declare piece array and initialise it
        piece = new double[length];
        randomizeValues();
};

//decon
vDNA::~vDNA() {
        //only need to deallocate space for piece array
        delete [] piece;
};

//copy functions
vDNA::vDNA(vDNA &copy) {
        //directly copy everything across
        for (int i=0; i<length; i++)
                piece[i] = copy.piece[i];
        fitness[0]=copy.fitness[0];
        fitness[1]=copy.fitness[1];
};
```

```
void vDNA::copyInto(vDNA *pDestDna) {
        for (int i=0; i<length; i++)
                pDestDna->piece[i]=piece[i];
        pDestDna->fitness[0]=fitness[0];
        pDestDna->fitness[1]=fitness[1];
}; //copyInto

//mutation
int vDNA::mutate(void) {
        //decide which gene to mutate
        int gene = randInt(maxHiddens)*geneLength;

        //decide which part of that gene to mutate and thus the offset
        int offset;
        int part = randInt(mutateTotal);
        if (part<mutateActive)
                //mutate active position
                offset=0;
        else //mutating a weight value
                offset=1+randInt(ins+outs);

        //decide on mutation type
        if (randDouble((double)1)<newValueOrGauss)
                { //use gaussian mutation
                        float stdDev=(float)0.1;
                        //then actually mutate
                        piece[gene+offset]=
                                        (double)newGaussian(piece[gene+offset],stdDev);
                }
                else //whole new value
                        piece[gene+offset]=(double)negPos();

        //must return success, useful having return for testing
        //mutation by returning mutation position when needed
        return 1;
}; //mutate

//crossover things
int vDNA::newCrossOverPoint(void) {
        //choose which node to have crossover point on
        int position = randInt(maxHiddens)*geneLength;
        //decide where on that node it's going to be
        int where = randInt(xTotal);
        if (where<xOverNodes)
                //crossover at active position
                return position;
        if (where<xOverNodes+xOverInsOuts)
                //crossover at first outgoing weight
                return position+ins+1;
        //crossover anywhere
        return position+randInt(geneLength);
}; //newCrossOverPoint

//breed based on crossover, int is number of crossover points
int vDNA::crossOver(vDNA *parentA, vDNA *parentB, int numPoints) {
        //make sure not too many crossover points!
        if (numPoints>=length)
                numPoints=length-1;

        //work out distinct crossover points
        int *crossOver = new int[numPoints]; //array for crossover points
        crossOver[0] = randInt(length);
        int which=1; //which crossover point we are deciding
        while (which!=numPoints) {
                //give a random value
                crossOver[which] = newCrossOverPoint();
                //check if it's distinct from previous ones
                int sameAs=false;       //assume not same as any other
                                                       //yet and show otherwise
                for (int j=0; j<which; j++)
                        if (crossOver[which]==crossOver[j])
                                sameAs=true;
                //if its not the same go to chossing next one
                if (!sameAs)
                        which++;
        };
        //define pChilds arrays
        int parentAToChild = 1; //ie parentA giving gene to pChild A
        for (int i=0; i<length; i++) {
                //is this a crossover point? compare against all crossovers
                int test=0;
                while (test<numPoints) {
                        if (i==crossOver[test]) {
                                //swap which parent coming from
                                parentAToChild = !parentAToChild;
                                test=numPoints;
                        };
                        test++;
                };

                //transfer actual dna strand information
                if (parentAToChild)
                        piece[i] = parentA->piece[i];
                else  //!parentAToChildA
                        piece[i] = parentB->piece[i];
        }; //for

        //free memory
        delete [] crossOver;
        //came out ok
        return 1;
};

//inversion things
//swap two genes, used during inversion
void vDNA::swap(int a,int b) {
        int i; //loop iterator
        double *temp=new double [geneLength]; //for temp storage
        //store gene 'a' temporarily
        for (i=0; i<geneLength; i++)
                temp[i]=piece[a*geneLength+i];
        //copy gene 'b' into gene 'a'
        for (i=0; i<geneLength; i++)
                piece[a*geneLength+i]=piece[b*geneLength+i];
        //copy temp stored gene into gene 'b'
        for (i=0; i<geneLength; i++)
                piece[b*geneLength+i]=temp[i];
        //free memory
        delete [] temp;
}; //swap

//cyclic inversion
void vDNA::inversion(void) {
        //pick distinct inversion positions
        int pt1=randInt(maxHiddens);
        int pt2=pt1; //set equal to force following loop at least once
        while (pt1==pt2)
                pt2=randInt(maxHiddens);
        //do actual inversion between pt1 & pt2
        while (pt1!=pt2) {
                //do one positions swap
                swap(pt1,pt2);
                //check to see if done by pts being within one of each other
                //either nornally, or in the boundary case
                if ((pt2-pt1==1)||(pt1-pt2==maxHiddens-1)) {
                        //force finish of loop
                        pt1=pt2;
                } //if
                else {
                        //not finished so move points closer, modulo maxHiddens
                        pt1++;  if (pt1==maxHiddens)   pt1=0;
                        pt2--;  if (pt2<0)                           pt2=maxHiddens-1;
                }; //else
        }; //while
}; //inversion

//for debuggging
void vDNA::display(int w) { //w=0, display all, w=1 display only fitness values
        if (w!=1)
                for (int i=0; i<length; i++) {
                        cout << piece[i] << " ";
                        if ((i+1)%geneLength==0) cout << endl;
                };
        cout << "rawFit=" << fitness[0] << " scaledFit=" << fitness[1] << " ";
        cout << "numHiddens=" << numHiddens() << endl;

        int y; cin >> y;
}; //display
```

Position header and class definitions

```
/*      POS.H
        data structure and procedures for 2d distances
        Matthew Kelcey
        Honours Research Code
*/

#ifndef POS_H
#define POS_H

#include <math.h> //for sqrt
#include <iostream.h>

class Position {
        friend ostream &operator<<(ostream &output, const Position &loc) {
                cout << "(" << loc.x << "," << loc.y << ")";
                return output;
        };
public:
        Position() { x=y=(double)0; };
        Position(double nx, double ny) { x=nx; y=ny; };
        inline double distTo(Position other) {
                return sqrt((x-other.x)*(x-other.x)+(y-other.y)*(y-other.y));
                };
        void set(double nx, double ny) { x=nx; y=ny; };
        double x,y;
};

#endif
```

Entity header file

```
/*      ENTITY.H
        structure for holding information about each entity
        by Matthew Kelcey
        Honours Research Code.
*/

#ifndef ENTITY_H
#define ENTITY_H

#include "virtuDNA.h"
#include "neural2.h"
#include "pos.h"
#include "colour.h"

const double turnAngle = 0.1; //radians
const double stepSize = 0.05; //size of world is 0->1
                                                    //(easier for scaling purposes)
//entity vision constants
const double pi = 3.14159265358;
const double twoPi=pi*2;
const double fieldOfView = (double)pi/2;//90deg
const double gamma = 0.5; //1=normal brightness, <1 more, >1 less.
const int fTracking = 1; //write vision to output vision.txt

enum eDisplay {locationInfo,visionInfo}; //for displaying entity values
enum eEntityType {leader, follower};


/******************************************************************

grid is laid out as....

  0,0          maxX,0         l = acw
                                                   r = ccw
        pi/2
     pi       0                            have to show eyes in reverse order.
        3pi/2

  0,maxY         maxX,maxY

 ******************************************************************/

class Entity {
public:
        //construct with null values
        Entity();
        //default decon
        ~Entity();
        //default entity from dna
        void constructFromDNA(vDNA*);
        //init all values
        void init(void);

        //snapshot vision
        void snapShotVision(Entity*); //give last entity in list
        //listen for sounds
        void listen(double[]); //noises from world

        //calculate the sweep angle to another position
        double sweepAngleTo(Position);
        //get entity to think, then move, then return the 'move'
        // eg 'n' (north) or 'l' (left) for possible displaying
        char move(eEntityType, //denote whether leader or not
                         double[]);   //world noise array for passing messages
        //display info on entity
        void display(void);           //display all information
        void display(eDisplay);       //location=all location spec variables

                                                   //vision=just the vision

        //put entity to a new location and give direction
        void relocate(Position,double);
        //change an entities colour
        void changeColour(Colour);
        //reset the raw fitness value
        void resetFitness(void);

        //entity variables
        //eEntityType  type;          //entity is animate or inanimate
        Network *pController;         //controller
        Entity *pNextEntity;          //used when applying vision calcs
        Position loc;                         //location
        double direction;                     //direction facing
        Colour looks;                         //colour of the entity
        vDNA *pSourceDna;                     //need this pointer to allocate fitness
        //the following 3 are dynamically defined (since inanimates dont use them)
        Colour *vision;                               //what the entity can see
        //these two are dynamic since messLength may =0
        double *hearing;                      //what the entity hears
        double *voice;                        //what the entity says
};
#endif
```

Entity class definition

```
/*      ENTITY.CPP
        class defn for entities
        last modified 4/5 for initial writing
*/

#include "entity.h"
#include "fstream.h"
#include <assert.h>
#include <math.h>             //for sin & cos
#include <iostream.h>  //for debugging info

void Entity::display(eDisplay locOrVision) {
        if (locOrVision==locationInfo) {
                cout << "entity is ";
                if (pController==NULL) cout << "in";
                cout << "animate @" << loc
                         << " f:" << (double)direction
                         << " colour is " << looks << endl;
        }
        else {//do vision
                cout << "can see ";
                for (int i=numEyes-1; i>-1; i--)
                        cout << i << ":" << vision[i] << "  ";
                if (numEyes==0)
                        cout << "nothing, this entity is blind";
                cout << endl;

                //do hearing
                cout << "can hear ";
                for (i=0; i<messLength; i++)
                        cout << i << ":" << hearing[i] << " ";
                if (messLength==0)
                        cout << "nothing, this entity is deaf";
                cout << endl;
        }; //vision and hearing
};

void Entity::display(void) {
        display(locationInfo);
        //dont bother with displaying vision for inanimates
        if (pController!=NULL) {
                display(visionInfo);
                cout << "controller is " << pSourceDna->ins << "x"
                         << pSourceDna->numHiddens() << "("
                         << pSourceDna->maxHiddens << ")x"
                         << pSourceDna->outs << endl;
        };
};

void Entity::init(void) {
        loc.set(0,0); direction=0;
        looks.set(0,0,0); int i;
        //construct vision array for the entity
        if (numEyes!=0)
                vision = new Colour[numEyes];
        //make it see nothing
        if (numEyes!=0)
                for (i=0; i<numEyes; i++)
                        vision[i].reset();
        //construct hearing array
        if (messLength!=0) {
                hearing = new double[messLength];
                voice   = new double[messLength];
        }; //if
        //make it hear nothing
        if (messLength!=0)
                for (i=0; i<messLength; i++)
                        hearing[i]=(double)0;
        //not using vision for dont bother maintaining list
        pNextEntity==NULL;
};
```

```
Entity::Entity(void) {
        //no controller for this entity yet
        pController = new Network();
        //no source dna yet either
        pSourceDna = NULL;
        //reset other values
        init();
};

void Entity::constructFromDNA(vDNA *pDna) {
        //make a controller from this dna
        pController = new Network(pDna);
        //remember where this dna came from
        pSourceDna=pDna;
        //ensure other values have been reset
        init();
};

Entity::~Entity() {
        //free up reserved memory space
        delete pController;
        //free vision array
        if (numEyes!=0)
                delete [] vision;
        //and hearing array
        if (messLength!=0) {
                delete [] hearing;
                delete [] voice;
        };
};

//sweep angle function used in a few other places also`
double Entity::sweepAngleTo(Position otherLoc) {

        //store the relevant variables
        double x1=loc.x;
        double y1=loc.y;
        double x2=otherLoc.x;
        double y2=otherLoc.y;

        //work out angle from axis between point 1 and 2
        double tAngle;
        //check for div by zero error
        if (x2==x1)
                if (y2<y1)      tAngle = -(double)pi/2;
                else tAngle = (double)pi/2;
        else
                tAngle = atan((double)(y2-y1)/(x2-x1));
        //quadrant 2 & 3
        if (x2<x1)              tAngle+=pi;
        //quadrant 4
        if (tAngle<0)  tAngle+=twoPi;

        //work out relative sweep angle
        tAngle-=direction;
        if (tAngle<0)           tAngle+=twoPi;
        if (tAngle>twoPi)       tAngle-=twoPi;

        return tAngle;
};

void Entity::snapShotVision(Entity *pCompareEntity) { //pIter==pFirstEntity[inanimate]
        //some needed variables
        double halfFOV = (double)fieldOfView/2;
        //RE int        numInVision[numEyes];  //number entities in each view
        int     fFinished=false; //flag to decide when finished

        //clear out num in vision array and clear vision
        for (int i=0; i<numEyes; i++) {
        //RE    numInVision[i]=0;
                vision[i].reset();
        };

        //compare with other entities in the entity list
        while (!fFinished) {
                //dont want to look at self
                if (pCompareEntity!=this){
                        //calculate the sweep angle to this entity we are comparing
                        double angle=sweepAngleTo(pCompareEntity->loc);

                        //check if its in the field of view
                        if (angle<halfFOV || angle>twoPi-halfFOV) {
                                //in view, but which eye?
                                //scale to -halfFOV -> halfFOV
                                if (angle>pi) angle-=twoPi;
                                //scale to 0->FOV (halfFOV-angle that is)
                                angle+=halfFOV;

                                //work out which eye its in then
                                int whichEye=(int)(angle/(double)(fieldOfView/numEyes));

                                //calc distance to the comparing entity
                                double dist = loc.distTo(pCompareEntity->loc) * gamma;

                                //add that sight to correct eye
                                vision[whichEye].red
                                        +=(double)pCompareEntity->looks.red / dist;
                                vision[whichEye].green
                                        +=(double)pCompareEntity->looks.green / dist;
                                vision[whichEye].blue
                                        +=(double)pCompareEntity->looks.blue / dist;
                                //keep record of how many in this view
                                //RE numInVision[whichEye]++;
                        } //if in view
                }; //if not looking at self

                //even newer version for just the one list
                if (pCompareEntity->pNextEntity!=NULL)
                        pCompareEntity = pCompareEntity->pNextEntity;
                else
                        fFinished=true;
        }; //while not finished flag loop

        //get ready to append if needed
        ofstream visionFile ("vision.txt",ios::app);

        if (fTracking)
                visionFile << numEyes << " " << endl;

        //average out vision values and put result in entity storage
        for (i=0; i<numEyes; i++)
                //average out what was seen in each eye
                if (numInVision[i]!=0) {
                        vision[i].red  /=(double)numInVision[i];
                        vision[i].green       /=(double)numInVision[i];
                        vision[i].blue /=(double)numInVision[i];
                };

                //if tracking then send to file vision.txt
                if (fTracking)
                        visionFile     << vision[i].red << " "
                                              << vision[i].green << " "
                                              << vision[i].blue << " " << endl;
};

void Entity::listen(double sounds[]) {
        //copy sounds into hearing
        for (int i=0; i<messLength; i++)
                hearing[i]=sounds[i];
};

char Entity::move(eEntityType type, double worldNoises[]) {
        double *inputs  = new double[pSourceDna->ins];
        double *outputs = new double[pSourceDna->outs];
        int i;  //general loop variable
        char returnVal;

        //need to prepare inputs for controller
        int upto=0; //which part of the input we are defining
        //set first position to be 1 for bias calcs
        inputs[upto++] = 1;
        //add positional information, if entity has a global
        //positioning system (gps set=2)
        if (gps!=0) {
```

```
		inputs[upto++] = loc.x;
		inputs[upto++] = loc.y;
	};
	//add vision information, if the entity has vision
	if (numEyes!=0) {
		//transfer to input array
		for (i=0; i<numEyes; i++) {
			inputs[upto++] = vision[i].red;
			inputs[upto++] = vision[i].green;
			inputs[upto++] = vision[i].blue;
		}; //for
	}; //if
	//add noises, if the entity is not deaf and is a follower
	if ((type==follower) && (messLength!=0)) {
		for (i=0; i<messLength; i++)
			inputs[upto++]=hearing[i];
	};

	//should have filled up all the slots now
	assert(upto==pSourceDna->ins);

	//think about things
	pController->propogate(inputs,outputs);

	//extract from output the noise it made
	//but only if this is the leader
	if (type==leader)
		for (i=0; i<messLength; i++)
			worldNoises[i]=outputs[i];

	//if it is the first entity (the leader) exit now
	if (type==leader)	{
		delete [] inputs;
		delete [] outputs;
		return 'x'; //x representing no move
	};

	//find which is highest of the outputs
	double highestValue=outputs[0];
	int highest=0;
	for (i=1; i<decisions; i++) {
		if (outputs[i]>highestValue) {
			highestValue=outputs[i];
			highest=i;
		}; //if
	}; //for

	//make that actual move, depends on how many decisions there are
	if (decisions==3) {	//left, right and straight ahead
		switch(highest) {
		case 0: //turn left
			direction += turnAngle;
			returnVal = 'l';
			break;
		case 1: //turn right
			direction -= turnAngle;
			returnVal = 'r';
			break;
		case 2: //go straight
			//not yet implemented
			exit(666);
			returnVal = 's';
			break;
		};
	}
	else {//should be four then!
		assert(decisions==4);
		switch (highest) {
		case 0: //north
			loc.y-=stepSize;
			returnVal = 'n';
			break;
		case 1: //south
			loc.y+=stepSize;
			returnVal = 's';
			break;
		case 2: //east
```

```
                        loc.x+=stepSize;
                        returnVal = 'e';
                        break;
                case 3: //west
                        loc.x-=stepSize;
                        returnVal = 'w';
                        break;
                }; //end of switch */
        }; //else

        //free inputs
        delete [] inputs;
        delete [] outputs;
        return returnVal;
        //ps. doing checking for world wrap by world object
};

void Entity::relocate(Position newLoc, double newDirection) {
        loc=newLoc;
        direction=newDirection;
};

void Entity::changeColour(Colour newColour) {
        looks=newColour;
};

void Entity::resetFitness(void) {
        pSourceDna->fitness[raw]=(float)0;
};
```

Colour header and class definitions

```
/*	colour structure for vision things
	Matthew Kelcey
	Honours Research Code
*/

#ifndef COLOUR_H
#define COLOUR_H

class Colour {
	friend ostream &operator<<(ostream &output, const Colour &c) {
		cout << "(" << c.red << "," << c.green << "," << c.blue << ")";
		return output;
	};
public:
	Colour() {red=green=blue=(double)0;};
	Colour(double r, double g, double b) {red=r;green=g;blue=b;};
	void set(double r, double g, double b) {red=r;green=g;blue=b;};
	void reset(void) {red=green=blue=0;};
	double red,green,blue;
};

#endif
```

Neural Network header file

```
/*      NEURAL2.H
        3-level neural network object class
        inputs and outputs in range -1 to 1
        no momentum implementation or flat spot weight correction

        by Matthew Kelcey
        Honours Research Code
*/

#ifndef NEURAL2_H
#define NEURAL2_H

#include <stdlib.h>           //for rand() function used in creation
#include "virtudna.h"

struct listNode {
        double *weights; //array to be dynamically created
        listNode *next;
};

class Network {
public:
        //default
        Network(void);
        //number nodes in layers (input,output), and dna;
        Network(vDNA*);
        //destuctor for removing dynamically created arrays
        ~Network();

        //progogate inputs through network and sets outputs
        void propogate(double[],double[]);
        //propogate and determine magnitude of error (tests & true values)
        double errorMagnitude(double[], double[]);
        //train network with input array and true values array and training rate
        void train(double[], double[], double);

        //make from the dna
        void constructFromDNA(vDNA*);
        //inject network info back into the dna
        void injectToDna(void);

        //raw fitness accessing
        void clearRawFitness(void);
        void addToRaw(double);

        //for debugging
        void display(void);

        //this should be private but I trust my own access to it
        int hiddenNodes; //number of

private:
        //keep pointer to parent dna for changing after training
        vDNA *parentDna;
        //need to store which part of the chromosone each node came from
        //for reinjecting trained values back into the actual dna
        int *positions;

        ////implement weight values in terms of linked list of arrays since
        ////it gave optimal performance under testing.
        //list of arrays containing weights in hidden layer
        listNode *hiddenWeights;
        //list of arrays containing weights in output layer
        listNode *outputWeights;
        //hidden layer node values (to be created dynamically)
        double *hiddenValues;
};

#endif
```

Neural Network class definition

```
/*      NEURAL2.CPP v3.0
        neural class method definitions
        by Matthew Kelcey
        Honours Research Code.
*/

#include "neural2.h"
#include <math.h> //for exp and pow
#include <iostream.h> // for debugging

void Network::addToRaw(double value) {
        parentDna->fitness[raw]+=value;
};

void Network::clearRawFitness(void) {
        parentDna->fitness[raw]=0;
};

//dot product function for fast double dp's
//optimised for pipelining on PentPro
double dotProd(int len, double *a, double *b) {
        int k,m;
        double sum=(double)0;
        k=len/4;
        m=len%4;
        while (k--) {
                sum += *a * *b;
                sum += *(a+1) * *(b+1);
                sum += *(a+2) * *(b+2);
                sum += *(a+3) * *(b+3);
                a += 4;
                b += 4;
        };
        while (m--)
                sum += *a++ * *b++;
        return sum;
};

//constructors
Network::Network(void) {
        //nothing to do yet, useful for allocating
        //space before actual dna is known
};

Network::Network(vDNA *dna) {
        //make it from the dna
        constructFromDNA(dna);
};

void Network::constructFromDNA(vDNA *dna) {
        int i; //loop variable

        //store dna pointer for changing when training
        parentDna=dna;
        //define the numbers of nodes in each layer
        hiddenNodes = dna->numHiddens();

        //define lists for weights for network from dna
        //and construct them now
        hiddenWeights = new listNode;
        outputWeights = new listNode;
        //and array for holding which genes nodes are drawn from
        positions = new int[hiddenNodes];

        //make list structure for hidden weights list
        listNode *iter;
        iter=hiddenWeights;
        for (i=0; i<hiddenNodes-1; i++) {
                iter->weights = new double[parentDna->ins];
                iter->next = new listNode;
                iter=iter->next;
        };
        iter->weights = new double[parentDna->ins];
        iter->next = 0;
        //and also make list for output weights arrays
        iter=outputWeights;
        for (i=0; i<parentDna->outs-1; i++) {
                iter->weights = new double[hiddenNodes];
                iter->next = new listNode;
                iter=iter->next;
        };
        iter->weights = new double[hiddenNodes];
        iter->next = 0;

        //extract data from strand and put it in these arrays
        //and also into bias values array, and store info on positions
        int upto=-1; //how far along dna strand we are
        listNode *iterH = hiddenWeights;
        listNode *iterO = outputWeights;
        int whichNode = 0; //which node we are adding now.
        //scan along strand and define weights for active nodes in hidden layer
        while (++upto < parentDna->length) {
                //is there a new array to define?
                if (dna->piece[upto] > 0) {
                        //store where it came from
                        positions[whichNode] = upto;
                        //get values for hidden layer
                        for (i=0; i<parentDna->ins; i++) //can put upto++ in here?
                                iterH->weights[i] = dna->piece[++upto];
                        iterH = iterH->next;

                        //get values for output layer
                        iterO = outputWeights;
                        while (iterO->next!=0) {
                                iterO->weights[whichNode] = dna->piece[++upto];
                                iterO = iterO->next;
                        };
                        iter->weights[whichNode] = dna->piece[++upto];
                        //up to adding potential next node
                        whichNode++;
                }
                else //skip along strand to next node definition
                        upto += parentDna->geneLength-1;
        };

        //create dynamic array for holding values propogated through network
        hiddenValues = new double[hiddenNodes];
};

//to delete a list, used by deconstructor
void deleteList(listNode *iter) {
        while (iter->next!=0) {
                delete [] iter->weights;
                iter=iter->next;
        };
        delete [] iter->weights;
}; //deleteList

//destroy the network by freeing memory used by the arrays
Network::~Network() {
        //free up memory
        deleteList(hiddenWeights);
        deleteList(outputWeights);
        delete [] hiddenValues;
        delete [] positions;
};

//propogate values from x array through network to y array
void Network::propogate(double inputs[],double output[]) {
        int i; //for loops

        //evaluate values for hidden nodes (with sigmoid function)
        listNode *iter = hiddenWeights;
        for (i=0; i<hiddenNodes; i++) {
                hiddenValues[i] = dotProd(parentDna->ins, inputs, iter->weights);
                hiddenValues[i] = (double)(1/(1+exp(-hiddenValues[i])));
                iter = iter->next;
        };
        //evaluate values for output nodes
        iter = outputWeights;
        for (i=0; i<parentDna->outs; i++) {
                output[i] = dotProd(hiddenNodes, hiddenValues, iter->weights);
                iter = iter->next;
        };
};

//propogate and determine magnitude of error
double Network::errorMagnitude(double input[], double trueValue[]) {
        //for comparing against true values
        double *output = new double[parentDna->outs];
        double errorMagnitude = (double)0;

        //get outputs first for error comparison;
        propogate(input,output);

        //calculate error magnitude as the mean square error
        for (int j=0; j<parentDna->outs; j++)
                errorMagnitude += (double)pow(trueValue[j]-output[j],2);
        errorMagnitude /= parentDna->outs;

        //free memory
        delete [] output;

        return errorMagnitude;
};

//train network byb modifying weights given
//input array inputs and true values array
void Network::train(double inputs[], double trueValues[],
                                                                double trainingRate) {
        int i,j,k; //for array handling
        //outputs for calculating error magnitudes
        double *outputs=new double[parentDna->outs];
        listNode *iterO, *iterH; //iterator for output nodes and hidden nodes

        //first propogate values through the network to obtain outputs
        propogate(inputs,outputs);

        //adjust hidden weights
        iterH = hiddenWeights;
        for (j=0; j<hiddenNodes; j++) {
                //do sumation
                double sum=(double)0;
                iterO = outputWeights;
                for (k=0; k<parentDna->outs; k++) {
                        sum += (outputs[k]-trueValues[k])*iterO->weights[j];
                        iterO=iterO->next;
                };
                for (i=0; i<parentDna->ins; i++)
                        iterH->weights[i]-=trainingRate*hiddenValues[j]*
                                                                (1-
hiddenValues[j])*inputs[i]*sum;
                iterH=iterH->next;
        };

        //adjust output weights
        iterO = outputWeights;
        for (k=0; k<parentDna->outs; k++) {
                for (j=0; j<hiddenNodes; j++)
                        iterO->weights[j]-=trainingRate*(outputs[k]-trueValues[k])

        *hiddenValues[j];
                iterO=iterO->next;
        };

        //free memory
        delete [] outputs;
};

void Network::injectToDna(void) {
        listNode *iterH = hiddenWeights;
        listNode *iterO = outputWeights;

        //go through each of hidden nodes writing back to dna
        for (int whichNode=0; whichNode<hiddenNodes; whichNode++) {
```

```
		//where to start injecting
		int upto=positions[whichNode]+1; //add one since not changing active
term
		//inject weights for hidden layer (incoming weights)
		for (int j=0; j<parentDna->ins; j++)
			parentDna->piece[upto++] = iterH->weights[j];
		iterH = iterH->next;
		//inject weights for output layer (outgoing weights)
		iterO = outputWeights;
		while (iterO->next!=0) {
			parentDna->piece[upto++] = iterO->weights[whichNode];
			iterO = iterO->next;
		}; //while
		parentDna->piece[upto++] = iterO->weights[whichNode];
	}; //for i
};

/// debugging routines

//for displaying weights arrays
void displayArray(double *array, int length) {
	for (int i=0; i<length; i++)
		cout << array[i] << " ";
	cout << endl;
};

void Network::display(void) {
	cout << "network is " << parentDna->ins << ", "
		 << hiddenNodes << ", " << parentDna->outs << endl;
	//display hidden node weights
	listNode *iter=hiddenWeights;
	cout << "HIDDEN WEIGHTS" << endl;
	while (iter->next!=0) {
		//print out array
		displayArray(iter->weights,parentDna->ins);
		iter = iter->next;
	};
	displayArray(iter->weights,parentDna->ins);

	//display output node weights
	iter=outputWeights;
	cout << "OUTPUT WEIGHTS" << endl;
	while (iter->next!=0) {
		//print out array
		displayArray(iter->weights,hiddenNodes);
		iter = iter->next;
	};
	displayArray(iter->weights,hiddenNodes);

	int q; cin >> q;
}; //displayWeights
```