AN INVESTIGATION INTO THE PERFORMANCE OF A GENETIC ALGORITHM FOR THE SELECTION OF NEURAL NETWORKS

Bachelor of Engineering
Fourth Year Project Report

By: Oliver Diessel
Supervisor: Bruce Penfold

University of Newcastle
November 1990

Acknowledgements

I would like to thank Bruce Penfold, my supervisor, and Michael Sculley for their assistance and comments in the course of this investigation.

I would also like to thank my employer, Allco for its support.

Table of Contents

1 Abstract

This project extends the work carried out by Michael Bentink in a 1989 project "Neural Networks, a Genetic Breeding Algorithm", which concerned itself with producing neural networks with natural abilities for performing particular tasks by combining structural features of above average networks using a genetic algorithm [1].

The performance of the algorithm, as defined by its capacity to select networks to perform a given task, was investigated in this project.

The investigation began with verification of Bentink's results before it focused on the mutation strategy used by the algorithm to improve its performance. With an improved mutation model, the algorithm was tested on the XOR task and a seven input parity bit generation task. The results, which compare favourably with those reported for other genetic algorithms and conventional neural network training algorithms, were presented at the IASTED International Symposium on A.I. and Neural Networks in Zürich, Switzerland in June, 1990 [2].

The search for optimum performance led to the identification of a higher order parameter space which governs the performance of the algorithm. Although some work has been carried out in this area by other researchers, noteably Kauffman & Levin [3], Baba [4] and Bartlet & Downs [5], there has been no systematic empirical investigation of the effects of parameter variation on the performance of a genetic algorithm used for the selection of neural networks. The main result of the investigation is that the performance of the algorithm is affected by variations in key parameters and that their influence is modified by the criterium used for evaluating fitness.

2 Introduction

There are three main areas of activity in neural network research. The first concerns itself with appropriate structures for neural networks, the second with how to train neural networks to perform a task, and the third with how to apply these results to solve a problem.

Genetic algorithms have recently emerged as a general purpose optimisation technique. As yet they have not been broadly applied, yet they show promise in optimisation problems where potential solutions to a problem can be represented as a linear list of solution characteristics. Even more recently they have been applied to the problem of optimising the connection weights of neural networks, which is a problem faced by neural network trainers. Rather than training a network to perform a task, genetic algorithms work by selecting a network which can perform the task. At present they are competing strategies for the provision of a network to solve a problem. In any competition it is natural to ask which of the competitors is best? Unfortunately, the answer to this question is still a long way off, however, it is worthwhile comparing the performance of the more common training algorithms with genetic algorithms as a first step.

Most training algorithms are based on a gradient descent optimisation strategy. Of these, by far the most widely applied is backpropagation. Backpropagation aims to minimise the mean square error between the actual output of a network and the expected output by relying on the continuous differentiability of the neural function. For a given error, the required change in connection weights can be inferred from a knowledge of how a node will treat its inputs. Hence modifications are propagated back through the network. An iteration of the algorithm is defined as the modification of all network weights by propagation of the error back through the network upon presentation of a member of the training set.

The approach is deterministic and reductionist - the algorithm requires detailed knowledge of the network structure and neural functions. It is a relatively easy approach to analyse, at least for small networks, and it is easy to appreciate that it will be fast to converge if there is a good slope in the error. However, if the error is not unimodal, backpropagation, and indeed any gradient descent method can get trapped by local optima in the error. It will be slow to

converge if the error is (almost) flat. The method is difficult to apply to networks containing feedback connections, and does not work with neural functions which are not differentiable. Another drawback is that the method does not scale well, i.e. the effort required to train a network grows faster than the increase in the size of the network. These problems aside, the algorithm may lay claim to many successes in the field, and this is probably why it is a favourite.

In contrast, a genetic algorithm is a stochastic optimisation technique. It also aims to minimise an error function defined by the difference between the actual and expected network outputs, but it need not be the mean square error: any L_p norm will do, since its purpose is only to rank a population of networks according to their ability to perform a task, or according to their fitness. By representing a neural network as a string of connection weights, the algorithm randomly recombines the strings of a pair of high fitness networks (called parents) to form a new network (called a child) which hopefully performs the given task better. To compensate for the fact that a crucial connection is not present in the parents, there is a finite probability that a connection is mutated when the child is formed. An iteration of the algorithm is defined as the generation and evaluation of a new (set of) child(ren) to decide whether to include the new individual(s) in the population or not.

The approach is non-deterministic and holistic - the algorithm is oblivious to the structure of the network and the neural function. It is difficult to analyse, even for small networks, yet it is easy to appreciate that it is powerful in terms of its ability to span the space of solutions defined by the network structure. The method can avoid convergence to non-optimal solutions since mutations offer the opportunity to investigate the entire solution space. Feedback connections are treated as any other connection, and so pose no problem. Genetic algorithms have also been shown to scale well. The difficulty in their application arises from the fact that they are probabilistic. Results are not reproducible in general, they are sensitive to variations in parameters such as the mutation variance, and because of their relatively recent entry to the field, there is not much information available on their implementation.

A brief review of the work carried out by Bentink will serve as motivation for the investigation carried out in this project.

Bentink ran a genetic algorithm on two tasks. The first, the selection of a network capable of performing the XOR function, is simple and well documented in the literature. It has become somewhat of a benchmark task in neural network research, not only because of its impact on the direction of investigations in the field, but also because many researchers report the performance of their training algorithms on this task. Bentink's results, although sketchy, indicated relatively good performance on this task: the algorithm was capable of selecting a solution within 300 generations, on average, when a bit error scoring method was used [1]. If an iteration of the genetic algorithm is comparable to an iteration of the backpropagation training algorithm, then the two algorithms could be said to have performed roughly equally on the XOR task, since Rumelhart *et al* reported 250 iterations were required to train a network to perform the XOR function using this method [6].

The second task attempted was more complicated, although still relatively simple when compared to the range of uses neural networks have been applied to. The aim was to select a network which could recognise points on a cubic equation. However, the algorithm failed to achieve this goal. To begin examining the reasons behind this failure, the approach used, and the philosophy behind it need to be examined.

Bentink comments that the number of neuron layers in the brain is relatively small (6-10 layers), although he notes that recurrency can cause extremely long feedback chains. He points out that the small number of layers accounts for the speed with which we respond to our environment, and that it indicates a limit to the number of layers required for complex computation [1].

On the basis of this anatomical fact, Bentink chose to use 4 layer networks with 12 nodes per layer, since inputs and outputs were to be represented as 12 bit binary numbers.

The networks were presented with an input representing a point in the domain of the function and were expected to respond with the function value at that point. Feedback to the genetic algorithm took the form of individual bit differences. This choice was inappropriate because the algorithm could not have distinguished between networks having low order errors and networks which had the same number of errors, but in more significant bits. Although the

performance of the algorithm on this task may have been affected by the choice of scoring method, the lack of convergence was probably due to the choice of network structure and genetic algorithm parameters, which were guided more by the restrictions of the implementation and the run-time environment than by a consideration of the information storage and processing requirements of the networks. It was worth attempting the task once using this approach, yet a broader foundation for the choice of network and genetic algorithm parameters needs to be found before investigating such tasks further.

Any training or selection algorithm will fail to provide a network which can perform a given task if the networks they operate on are not complex enough to perform the task. The problem of finding an adequate structure falls into the first and third areas of activity of neural network research. This investigation sets out with the assumption that the networks which the genetic algorithm is expected to operate on have sufficient neurons to be capable of performing the given task. Although this approach does not necessarily lead to minimal networks, one of the degrees of freedom in the search for improved performance is eliminated. The investigation can therefore focus on the algorithmic parameters which affect performance.

There are two good ways to begin the search for improved performance. One way is to approach the problem from a theoretical point of view to begin with, before testing deductions and predictions via simulations. The other is to begin with an empirical approach to get a better feel for the problem before attempting to explain the phenomena observed from a theoretical point of view. The latter approach was adopted in this investigation. The advantages of using this approach are: it is less restrictive, because not so many simplifying assumptions need to be made; it is more expedient and convenient given the time available; and it is presumably easier to draw some conclusions or make broad deductions. Regrettably, this report contains more deductions than theoretical explanations.

Section 3 of this report describes the current implementation of the algorithm and highlights the differences between the methods adopted for this investigation and other methods commonly in use. Section 4 presents and discusses the results obtained in the course of the investigation. Section 5 identifies areas for further investigation and proposes some feasible

alternative implementations. Section 6 concludes the report by summarising the findings.

Appendix A of the report presents an overview of genetic algorithm concepts in the framework of a simple game playing example, and Appendix B contains a current listing of the program used in the course of the investigation.

Much has been written about what neural networks are and how they are used. It would be difficult to improve on the work done by Lippmann [7], Vemuri [8], Lapedes & Farber [9], and Palmer [10] to explain what forms their structures may take, how they are trained, and how they may be applied to the solution of practical problems. To this end, these topics will not be covered in this report, and the reader is directed to any one of these articles if he or she is unfamiliar with the field. Readers who are unfamiliar with genetic algorithms may find the information contained in Appendix A useful, as it introduces the terms and concepts used in the report.

3 Description of the Model

3.1 The Network Structure

The structure of the networks generated by the genetic algorithm is relatively unconstrained. They may be viewed as rectangular meshes of nodes, in which any interconnection between nodes is permitted providing no more than one input to a node originates from any particular node or external input.

3.1.1 Nodes

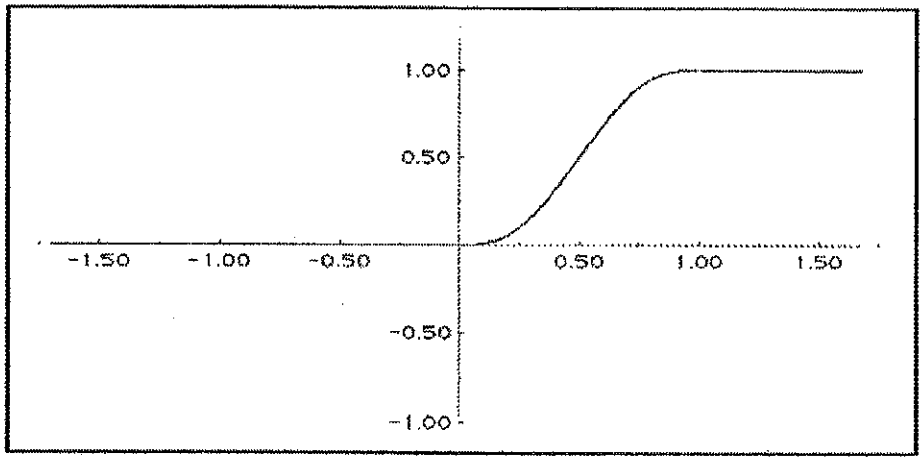
In keeping with traditional neural network models, the network nodes perform a non-linear function on the weighted sum of their inputs. All inputs are "transmitted" to a node via a connection having a particular weight or gain. All weighted inputs are summed by the node, and the result is passed through a sigmoid non-linearity with saturation. The sigmoid non-linearity causes the node to output 0 if the weighted sum is less than or equal to 0 and 1 if the weighted sum is greater than or equal to 1, otherwise it computes the output

$$F(s) := 6.4\,s^5 - 16\,s^4 + 10.8\,s^3 - 0.2\,s^2$$

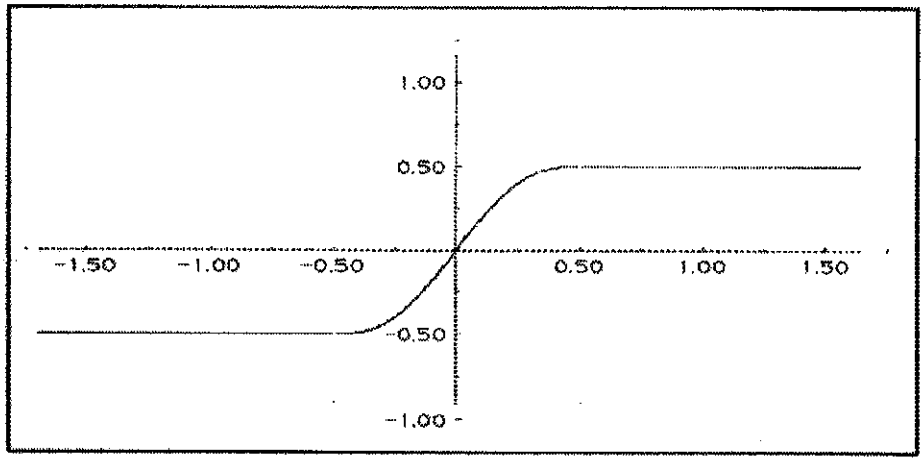
for the weighted sum, s.

The function is symmetric about the input and output values of 0.5.

The characteristic is atypical because in most implementations the sigmoid non-linearity is centred around 0. Also, it is unusual to have the response offset from 0 rather than some negative saturation value. That is, typically, the sigmoid non-linearity is chosen to saturate at an output of -0.5 for extremely negative inputs, and range up to 0.5, say, for extremely positive inputs. In this model, outputs are constrained to be positive, whereas more generally this is not the case. These differences are illustrated in Figure 3.1.



Sigmoid Function implemented



Sigmoid Function typically used

Figure 3.1 - Alternative Sigmoid Non-Linearity Representations

Note that a genetic algorithm does not require the neural function to be continuously differentiable, since it does not depend on a gradient descent method to optimise connection weights or biases. Thus a neural function represented as a step non-linearity is quite acceptable. Genetic algorithms therefore place less restrictions on the neural function than most training algorithms.

Any node may have a bias applied to it. The action of the bias may be viewed to be that of shifting the inflexion of the sigmoid function in a direction which opposes the sign of the bias. Alternatively, it may be viewed as another term of the weighted sum. Thus, a bias of 0.5 could be viewed as shifting the inflexion towards 0, or it could be viewed as adding 0.5 to the weighted sum of the node inputs. In either case, if the weighted sum of the inputs without the bias is less than 1.0, the bias will act to increase the output.

The number of nodes in the networks generated is controlled by two parameters : the number of nodes per layer and the number of layers in the networks, both of which are constant for a run. The current formulation requires that the number of nodes per layer be equal to the number of inputs to the network. The number of layers specified for the networks is guided by a consideration of the task to be performed. For example, in a classification task, the number of separating hyperplanes required to classify the input into the desired number of output classes determines the number of nodes or layers required.

Output is obtained from a designated node or set of nodes in the output layer.

3.1.2 Connections

The types of connections allowed between nodes, and between inputs and nodes, is one of the strongest features of this model, since the nodes may be interconnected to form a fully connected graph. Two way connections between nodes may be formed by having the output of one node inputting to a second, and vice versa. Nodes may connect with themselves or with no other nodes. Direct connections between the external inputs and the output layer may be formed as well. Thus the interconnections allowed are quite general: the networks may display feedback connections to earlier layers; feedforward connections to subsequent layers; and horizontal connections between nodes within a layer.

The weight of each connection is independent. When the networks are first generated, the networks are connected with a given probability of connection. If the procedure to generate these random networks decides a connection should be established between 2 nodes, or an

external input and a node, the weight will be assigned by a uniform random variable ranging in value between -1.0 and 1.0.

3.2 Computing with the nets

3.2.1 Representation

A network may be represented by a matrix whose elements specify the connection weights. This representation facilitates testing of the network, since the determination of a new network state is equivalent to multiplication of a vector representing the old network state by the weight matrix.

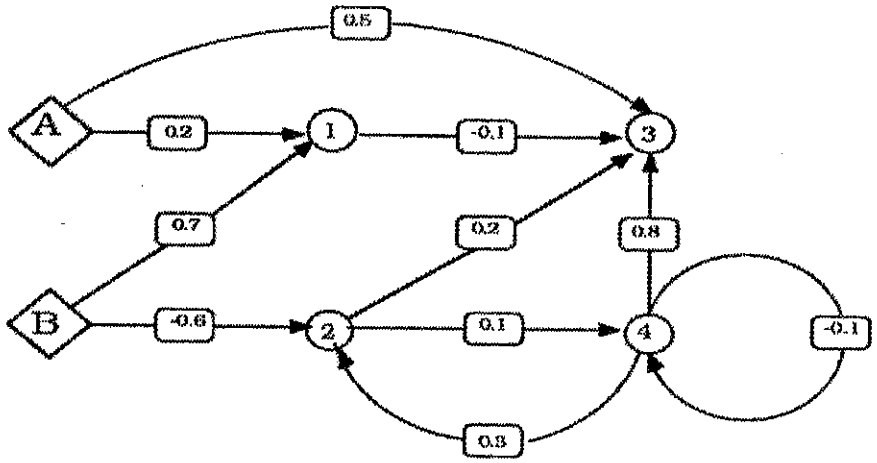
To achieve this, the matrix is defined as in Figure 3.2, which illustrates the weight matrix for a two input two layer XOR network. The column entries are the weights associated with connections from external inputs or node outputs whilst the row entries are the weights associated with node inputs. As Row 1 is scanned from left to right the weights associated with all inputs to node 1 may be read off. Similarly, as Column 3 is scanned, all outputs associated with Node 1 may be read off.

3.2.2 Testing

The network is tested by repeatedly multiplying a network state vector by the weight matrix until a specified number of multiplications has occurred, or until the state vector stabilises.

A state vector, X, is formed from leading entries corresponding to the input state being tested, and trailing entries equal to 0, the assumed initial state of all nodes. During the computation, the entries corresponding to the external inputs are held fixed, whilst the node state entries are updated at the completion of the matrix multiplication. The weighted sum of inputs, $s_i = \Sigma_j W_{ij} \cdot X_j$, is determined for each node in the network by calculating the usual matrix product for each row in the weight matrix, W. The bias applicable to node i is added to the weighted sum, before passing the weighted sum through the sigmoid non-linearity. The next state of the node, $x_i' = F(s_i)$ is then calculated. A new state vector, X′, is formed with the same

The two input two layer XOR network has the matrix representation W shown below.



Key to network:

(1) Node [0.2] Connection Weight ⟨A⟩ External Input

FROM

	input		layer 1		layer2			
	A	B	1	2	3	4		
W =	0.2	0.7	0	0	0	0	1	layer 1
	0	-0.6	0	0	0	0.3	2	
	0.5	0	-0.1	0.2	0	0.8	3	layer 2
	0	0	0	0.1	0	-0.1	4	

TO

Figure 3.2 - XOR Network Representation

leading (input) entries as X, but with updated node state entries. Computation proceeds until the state vector stabilises, or until a specified maximum number of cycles (multiplications) has transpired. Cycling is allowed for because of the instabilities which may arise from the feedback connections in the network.

The network is tested with each input condition for the task. The output may be passed through a hard-limiter centred around 0.5 to obtain a binary output, or used directly if real outputs are desired. Assuming the outputs of the network are unweighted, binary outputs are scored on the basis of the number of bit differences between the actual outputs and the expected outputs, whereas real outputs are scored on the basis of the mean square error between the actual and the expected outputs of the network.

A raw score for the network is obtained from the output performance of the network. The raw score is calculated as a fraction of the maximum (number of) error(s) possible. In addition, a measure of the network stability is obtained by measuring the percentage change in node states during the last test cycle for each input condition. This stability score is added to the raw score to obtain a final score for the network. Without considering the efficiency of the final network structure, a network which performs the given task without error and achieves stability during testing is called perfect.

3.3 Training

The networks are not currently being trained in the usual sense. The networks are being optimised to perform a task by selection.

3.4 The Genetic Algorithm

3.4.1 The Genetic Structure

To implement the genetic algorithm, a genetic structure which could describe the network structure had to be chosen. The aim of Bentink's project was to identify neural network structures with natural abilities. He therefore found it desirable to choose a gene which passed

on significant structural information from parent to child. The gene which was decided upon describes all connections from a layer in the network to a node in the network, realised as a list of connection weight values. Since the gene is atomic during the breeding process, this choice imparts more structural information onto the children than the choice of a single connection. There are natural alternatives to the chosen structure, as can be seen in Figure 3.3, however, the performance of the algorithm with these alternatives has not been investigated.

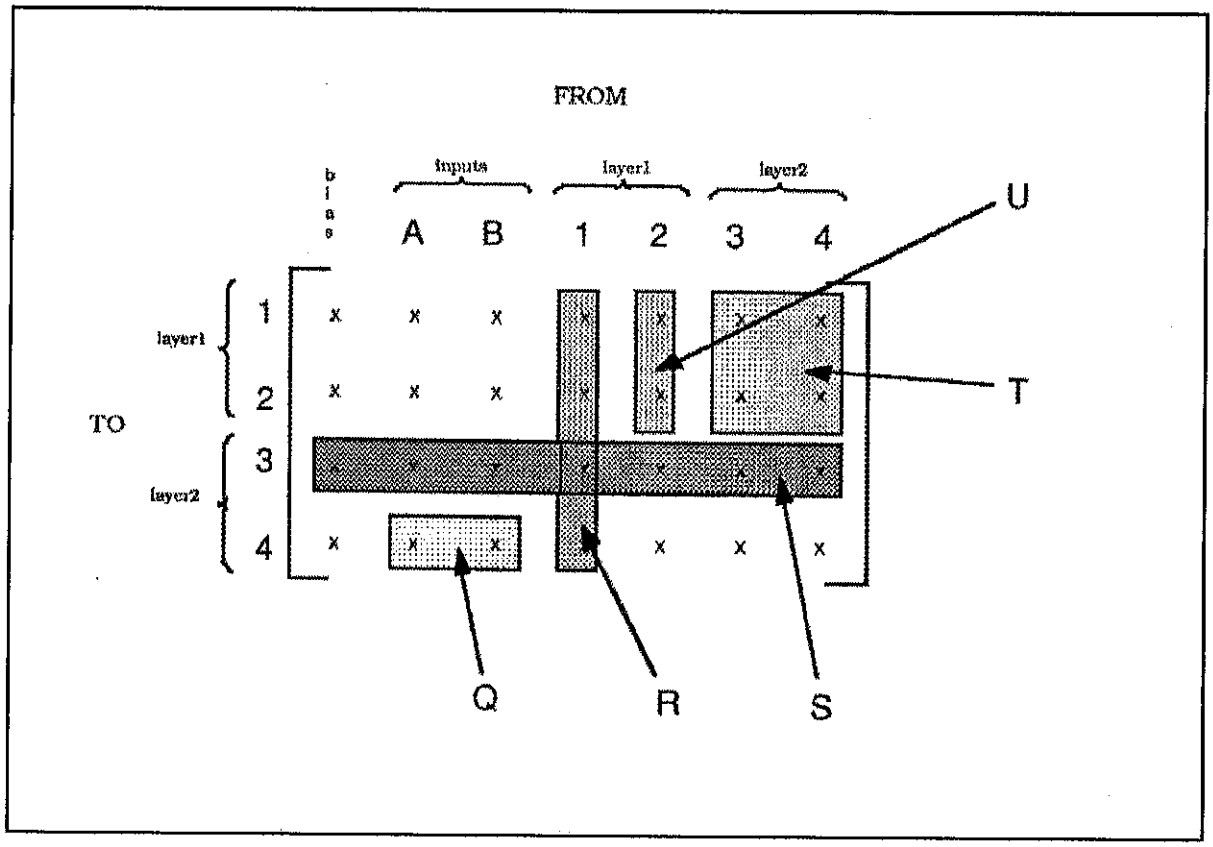


Figure 3.3 - Alternative Gene Representations

In Figure 3.3:

Structure Q represents all inputs to a node from a layer
Structure R represents all outputs from a node
Structure S represents all inputs to a node
Structure T represents all outputs from a layer to a layer
Structure U represents all outputs from a node to a layer

Having chosen the gene to be all connections from a layer to a node, a set of genes, called a gene segment, is needed to describe the interconnectivity of a node in the network because it may connect with all layers in the network. In addition to specifying a gene to describe the connections a node may have with the external inputs, the bias associated with the node is assigned a gene. Hence the number of genes needed to describe the node is two more than the number of layers in the network. Since the node may connect with all nodes in a layer, all genes but the bias gene contain as many connection weight entries as there are nodes in a layer. The bias gene consists only of the bias.

In this way, a network can be represented as a list of genes, called a gene string, to facilitate breeding. Gene strings are formed by concatenating the gene segments for each node in the network.

3.4.2 The Breeding Cycle

The breeding cycle commences with the generation of a specified number of random (unorganised) networks whose degree of connection, or connectivity, is given by a specified probability of connection. The connectivity is defined as the expressed fraction of all possible connections. The connection weights are determined by a uniform random variable ranging between -1.0 and 1.0.

This approach differs from Bentink's, who for the sake of gene pool diversity, decided the network designer should provide at least some of the networks for the original generation. Apart from this approach being more complicated, it biases the first few generations towards structures prescribed by the network designer. Although this approach may be of benefit when the networks are small, or when the designer has some idea of what the optimal structure should be, the power of non-determinism provided by the generation of random networks will be lost if the designer provides structures which can never be optimal, and the specified nets are not supplemented by random networks. In this case, the genetic algorithm needs to rely on mutations to achieve optimality. The possibility of optimising known solutions to tasks could be investigated using this approach, however, the efficiency of the algorithm decreases

as the optimality of the solutions increases, especially when optimal solutions are sparse, therefore it may not be particularly useful.

New networks are formed from each possible pair of networks in the population. Each parent pair defines a pair of children by specifying the contents of each gene in the child gene strings. The first child is defined by choosing a gene from the two parent genes with equal probability at each gene location. The second child is defined by assigning to it the parent gene not chosen by the first child, thus it may be viewed as the complement of the first child. The structures of the resulting children are therefore a mixture of the structures of the parents.

The method more commonly adopted in the literature is to split the parent genes at some random position along the gene string and to recombine the complementary fragments to form a child [4],[5],[11],[12],[13],[14],[15],[16]. The method used in this investigation is more powerful since it is possible for recombination of this form to occur, whereas the alternative method requires at least 2 generations to form a child composed of an initial and final section originating from one parent, and a middle section originating from another. The features of the two methods are illustrated in Figure 3.4.

3.4.3 Mutation

In addition to producing a pair of children for every pair of parents, the algorithm generates a mutated copy of each parent during the breeding cycle. Mutations are formed by making a copy of the parent and adding a normally distributed real random number centred at 0 to each connection of targeted genes. The severity of the mutation applied to each connection in a gene is controlled by specifying the variance of the mutation variable. The connection weight is kept within the range -1.0 to 1.0 by reflecting resulting connection weights which exceed this range back into it.

The number of genes targeted for mutation is determined by the degree of stability in the population, as measured by the change in average population fitness, and the expected fitness improvement of the best network, measured as the difference in fitness of the best networks of successive generations. High stability and poor improvement lead to a

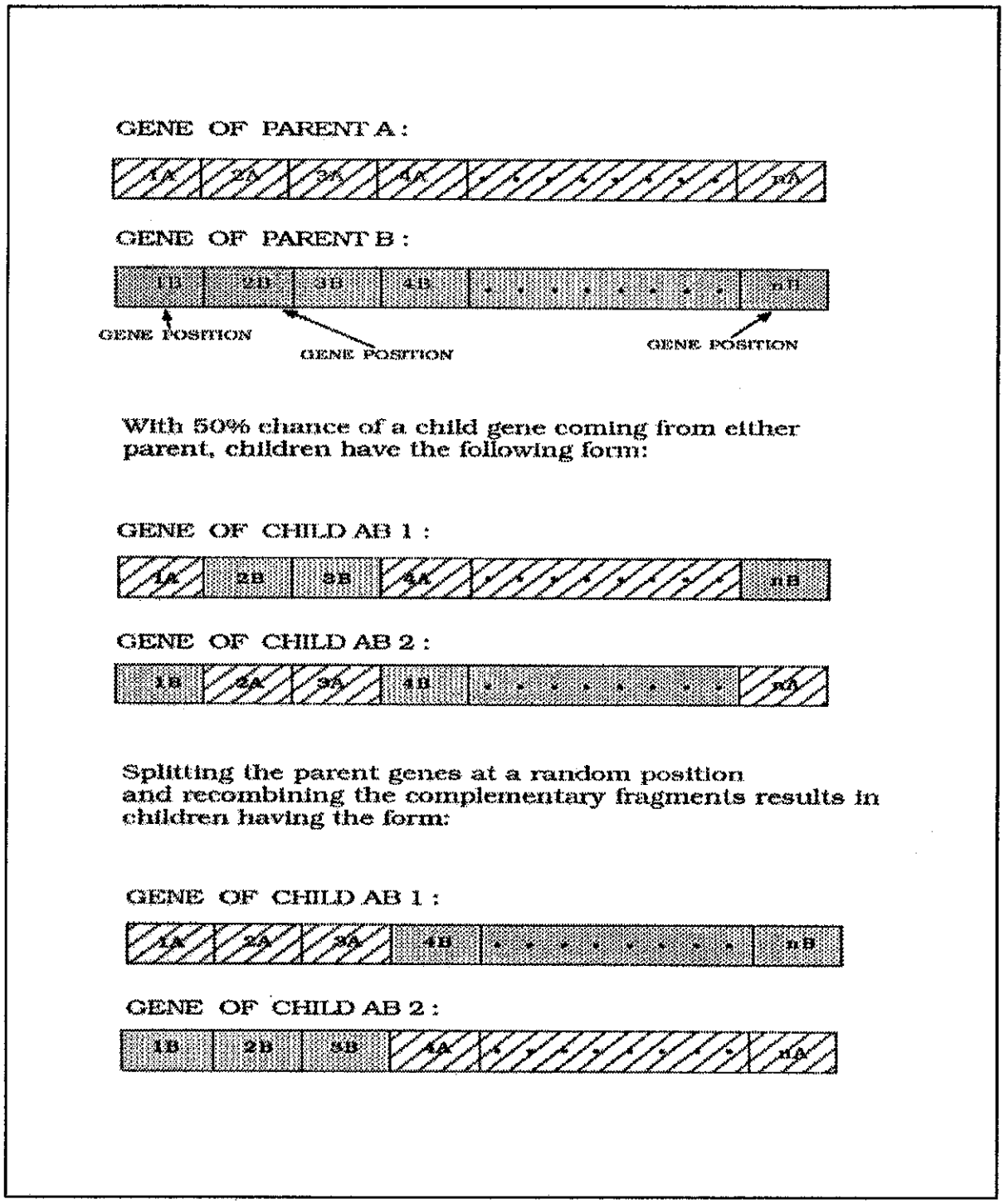


Figure 3.4 - Alternative Breeding Approaches

linear increase in the number of genes to be mutated, whereas the converse leads to a linear decrease. Genes to be mutated are selected by uniform random selection. The increment in the number of genes to be mutated is user definable.

This differs from typical genetic algorithms, where mutation takes the form of copy errors in the breeding process and is often realised as the addition of Gaussian noise to some of the connection weights.

3.4.4 Generation

The algorithm generates a pair of children for each possible couple and one mutant for each individual in the population. Bentink's model proceeded to test all individuals in the resulting population and select a specified number of networks to become the parents of the next generation. If there were N individuals before breeding, N^2 individuals were produced during breeding, resulting in a final population size of $N^2 + N$.

For large networks, this sequential implementation of the breed, test and select phases of the genetic algorithm severely restricted the population size if all networks were to be kept in the designers working set. To this end the model has been modified to test children and mutants as they are produced. Only the best N children are kept during breeding so that the maximum number of networks stored is 2N. After merging the parent and child lists, the worst N are discarded, leaving N parents for the next generation.

It should be noted that there is another significant difference between this implementation of the genetic algorithm and genetic algorithms documented in the literature [5],[15],[16]. This difference arises from forming all possible parent couples to breed children whereby all parents are replaced by children if N of them perform better than the most fit parent. The definition of a generation in this model is one breeding cycle as described above. In general, the algorithms encountered in the literature adopt the following approach: from the population, a pair of parents is chosen at random to produce a child. The parents are chosen with exponentially increasing bias towards the most fit end of the fitness spectrum. If the child produced is more fit than the least fit member of the population, then the least fit member

will be replaced by the child, otherwise, the child will be discarded. Although no reference has been found which defines a generation under this scheme, it is assumed that the process of selecting parents, breeding a child, testing it, and replacing another member of the population, or discarding it, is one generation.

In both schemes, an individual's fitness determines the probability of the individual's survival and therefore its expected number of offspring.

4 Results and Discussion

4.1 A Note on Convergence

In this report, convergence is said to have been achieved when a network is selected which performs the required task without error on the basis of the raw score. Note that this does not mean the network is expected to be stable, so the final score may still be non-zero due to the stability score.

When binary outputs are expected, performing the task without error is equivalent to generating the required binary outputs for all input cases. When real outputs are expected, convergence is assumed to have occurred when the final score (mean square error plus stability score) is less than the error which would have been generated by a single bit error had the binary scoring method been used. This definition allows the final scores of networks scored using either method to be directly compared for convergence.

The network designer needs to decide when a run should be abandoned. All runs reported in this report were abandoned after 2000 generations if perfection had not been achieved by then.

4.2 Rerunning the XOR Task

Bentink's report suffers from a lack of information about the run time parameters used to obtain results, and on the experimental results obtained. It was decided that his program for the XOR task would be rerun to obtain this data, and to proceed from there.

To this end, a copy of the program was obtained and converted to compile on a PC using Borland's TurboC. It was intended to carry out development work on a PC and conduct runs on an Apollo Prism.

Once the converted program was ported to the Apollo and recompiled, it was tested on the XOR task 10 times using an integer random number seed ranging from 1 to 10. In each run,

the probability of connection (connectivity) was set to 40%, and a maximum population size of 20 networks was specified. The program was required to generate a two input two layer solution. Generated networks were tested using the bit error method.

Of the 10 runs conducted, all but one failed to converge to a solution within 2000 generations. When it did converge, it did so at generation 375.

This result demonstrated that the program was able to select a network which could perform the XOR task, however, it did so with limited success. When the program was run, it was assumed there were only two differences between the runs conducted and those reported previously. The first was that a population size of 20 was specified rather than 10. And the second was that no initial networks were supplied, so the algorithm was expected to work on an entirely random initial population. It was expected that convergence would occur more rapidly and more often with the choice of a larger population size, however, Bentink's results could not be achieved.

It was subsequently noticed that a bug had been introduced during the conversion process. If the network contains more output layer nodes than required, the unused output layer nodes should be treated as "don't care" outputs. The bug caused the program to require these "don't care" outputs to be 0 for all input cases, thereby complicating the task the network was expected to perform. However, the original program was not rerun with this bug removed as explained below.

4.3 Approach to Mutation

To improve the performance of the algorithm, it was decided to focus on the approach taken to mutation. Originally, genes were targeted for mutation if a normal random variable (μ = 0.5, σ^2 = 0.2) was less than the "severity of mutation", a number based on the improvement in the fitness of the best network and the average fitness of the population over the previous generation. Targeted genes were modified by multiplying the individual connections of the gene by a normal random variable or by breaking or establishing connections with varying

probabilities. It was felt that apart from this approach being cumbersome, insufficient mutation was occurring.

To improve the situation, it was decided to linearly increase the number of genes targeted for mutation if the fitness of the best network did not improve sufficiently, or the average fitness of the population (a measure of genetic stability) did not change sufficiently during the previous generation. Conversely, if sufficient improvement did occur, and the population did not stagnate, the number of genes targeted for mutation would decrease linearly. The target genes would be selected by uniform random selection, and the mutation would be applied by assigning a new normally distributed weight ($\mu = 0.0$, $\sigma^2 = 0.5$) to each connection of targeted genes. This choice was inspired by the observation that Bentink's program performed better when the connection weights were assigned a new value, rather than being modified.

Using this approach, the network designer specifies the expected improvement in fitness per generation, the improvement requirement, and the expected difference in average fitness of the population between generations, the stagnation threshold. These factors affect the rate at which mutation is applied to the gene string. The designer also specifies the increment in the fraction of the gene string to be mutated if the expected fitness improvement and stagnation threshold are not exceeded (the mutation increment). This factor defines the granularity of mutation application by specifying how much more of the gene string is to be mutated with successive generations of stability. In addition, the designer may specify a base mutation fraction which is the minimum fraction of the gene string to be mutated.

The program was rerun with the following parameters:

Number of runs	10
Population size	20
Connectivity	0.4
Mutation increment	0.1
Improvement requirement	0.01
Stagnation threshold	0.01
Base mutation fraction	0.0

The results appear listed in Table 4.1.

The program was found to perform considerably better, with 7 out of the 10 runs converging. For those runs which converged, the average generation at which convergence occurred was found to be 573.

Random Number Seed	Generation at which convergence occurred
1	>2000
2	399
3	1178
4	874
5	>2000
6	308
7	>2000
8	299
9	166
10	785

Table 4.1 - Linear increase in mutation, new weight assigned.

The more usual approach taken by genetic algorithms is to add some Gaussian noise to genes targeted for mutation rather than assigning new values to the connections of targeted genes. This approach was implemented next, with the designer being allowed to specify the variance of the noise to be added. The procedure to generate random networks was also modified to produce networks having connections determined by a uniform random variable ranging between -1.0 and 1.0 rather than a normal random variable ($\mu = 0.0$, $\sigma^2 = 0.5$). This was done because it does not seem reasonable to bias the connection weights towards 0 given that a general network design procedure is sought. It also presupposes that the desired structure should have connection weights biased towards 0.

The program was rerun with the following parameters :

Number of runs	10
Population size	20
Connectivity	0.4
Mutation increment	0.1
Mutation variance	0.2
Improvement requirement	0.01
Stagnation threshold	0.01
Base mutation fraction	0.0

The results appear listed in Table 4.2.

Random Number Seed	Generation at which convergence occurred
1	>2000
2	227
3	1209
4	161
5	>2000
6	>2000
7	393
8	>2000
9	1288
10	771

Table 4.2 - Linear increase in mutation, Gaussian noise added.

The program was found to perform slightly worse overall, with 6 out of the 10 runs converging. For those runs which converged, the average generation at which convergence occurred was found to be 675.

Although the performance of the algorithm could be said to have decreased with this approach, the runs cannot be compared for the following reasons. For any random number seed, the initial population generated would have been quite different as a result of the new

definition for generating random networks. The sequences of random numbers generated to breed and mutate children would also have been quite different. As the sample size of 10 runs is possibly too small to resolve differences in performance, the relative merits of the two approaches need to be argued from a theoretical basis.

If the connection is assigned a new weight, it can be likened to a severe mutation. This has advantages when the task is unimodal (possesses a unique optimum) and in the initial stages of the run, when the algorithm randomly samples the solution space trying to find a rough solution [3]. When a solution is found, large discontinuities in connection weights are less likely to drive the network to a more optimal solution. Therefore, one would expect initially rapid convergence, followed by an exponential increase in the time required to achieve optimality, given the ever smaller changes in connection weights required to locate the optimum. The same problem can occur with the second method if the variance is too large, however, for small variance, one can expect asymptotic convergence to a local optimum [4]. Ideally, a compromise between early rapid convergence and late fine tuning of solutions is desired. This cannot be achieved by assigning a new weight but it might be achieved by reducing the variance as optimality is approached. Kauffman & Levin suggest applying a range of variances at every generation [3]. Thus in the early stages of the run, rough approximations to the solution should easily be found, and tuning is catered for by having small mutation variance. Final convergence to non-global optima is avoided by continuing to sample solutions a long distance away in the solution space with large mutation variance.

Although adding Gaussian noise of a specified variance goes someway towards this solution, Kauffman's approach has not been tested.

A final modification to the algorithm was carried out when it was noticed that the program was testing all output layer nodes rather than just the node(s) designated for output (see Section 4.2).

When the program was rerun with the same parameters as in the previous run, the results of Table 4.3 were obtained. As may be seen from the table, all runs converged, with the average number of generations required to achieve convergence being 121.

It may be assumed that the presence of the bug would have had a similar effect on all previous runs. Since the results of Table 4.2 indicate a better performance than the run using Bentink's original program, it can be assumed that, although the original program may have performed better with the bug removed, it would probably not have performed as well as the results listed in Table 4.3. The average number of generations required to achieve convergence was 129, which compares favourably with Bentink's claim that most runs "took 300 generations to find a suitable network" [1, p. 33]. However, Bentink did not report how many runs were made, nor how a suitable network was defined. The approach to mutation used here seems to result in better average performance of the algorithm.

Random Number Seed	Generation at which convergence occurred
1	129
2	508
3	4
4	5
5	329
6	14
7	56
8	27
9	30
10	103

Table 4.3 - Linear increase in mutation, Gaussian noise added, Output bug fixed.

Note that there is considerable variability in all results. This is probably due to the dependence of the algorithm on beneficial sequences of random events. In the results of Table 4.3, in runs 3 and 4 for example, it is obvious that a good initial population was generated, or rather, good genes were generated initially, and it took only a few generations to combine them in a way that produced a network which could perform the task. Runs 2 and 5 depict

the bad end of the scale, where the generation of poor genes in the original population meant the algorithm depended on mutations to improve the solutions derived from the initial set of genes and that the mutations were usually not good. Mutations are by nature random, and it can take some time before a suitable mutation occurs. It is more likely that a good chain of mutations is required to make a solution possible.

4.4 Comparison with other Algorithms

4.4.1 Stochastic Methods

The algorithm used selected networks which could perform the XOR task in an average of 129 generations. Bartlett & Downs report their genetic algorithm took between 200 and 1900 generations with an average of 680 generations to select networks which could perform the task [5]. The differences between their algorithm and the one used here are:

1. Bartlett & Downs use individual connection weights as genes.

2. They use the conventional genetic crossover operator during breeding.

3. They select breeding candidates with an exponentially decreasing likelihood from a parent list ranked by fitness.

4. Their networks differ structurally, as they do not include feedback connections.

5. Their testing method is different as well. Networks are tested using mean square errors rather than bit differences.

Matyas' random optimisation technique (a stochastic gradient descent optimisation method) relies solely on a mutation like operator to vary connection weights [4]. Although the algorithm is guaranteed to converge to the optimum solution asymptotically, it is not guaranteed to do so quickly. It was investigated for the XOR task and found to suffer from convergence problems for mutation variance in the range 0.001 to 1.2 [5]. Convergence

problems in this case means that no solution was found within the 5000 input presentations allowed. When it did converge, it typically required between 100 and 200 presentations.

4.4.2 Gradient Descent Methods

Rumelhart *et al* report that backpropagation requires 250 presentations to achieve convergence on the XOR task [6]. Pineda reports doing slightly better than Rumelhart, using his recurrent backpropagation training algorithm, achieving convergence after 200 presentations [17].

4.5 Seven Input Odd Parity Task

A second task was investigated to compare the performance of the algorithm against other methods. Bartlett & Downs report the performance of their algorithm on the selection of a network which can generate the odd-parity bit for a subset of all 7 bit words. The set they used appears in Table 4.4.

Input Bits							Output Bit
1	2	3	4	5	6	7	1
0	0	0	0	0	0	0	1
0	1	0	0	0	0	0	0
0	0	0	0	0	1	0	0
1	0	1	0	0	0	0	1
0	0	0	0	1	0	1	1
0	1	0	1	0	0	0	1
0	0	0	1	0	1	0	1
1	0	1	0	1	0	0	0
0	0	1	0	1	0	1	0
1	1	1	1	1	1	1	0

Table 4.4 - Incompletely Specified 7 Input Odd Parity Generator

Since this task, like the XOR task, falls into a class of tasks having enumerated input/output conditions, the program was generalised to perform this class of task. To implement and run any one of these tasks, the network designer needs to modify the network definition file, in order to specify the desired network structure, and the task definition file, in order to specify the appropriate output for each input.

The program was run 10 times with a population of 10 three layer seven input networks having 50% connectivity. The mutation variance was set to 0.01 and the mutation increment used was 0.04 (4.2 genes). The results obtained appear in Table 4.5.

Random Number Seed	Generation at which convergence occurred
1	>2000
2	>2000
3	>2000
4	>2000
5	982
6	467
7	7
8	549
9	479
10	370

Table 4.5 - Incomplete Seven Input Odd Parity Task results

For the runs which converged within 2000 generations, the average convergence occurred within 475 generations. This is much more than for the XOR task but is to be expected with the decrease in population size and increase in network complexity. The population size determines how many possible variants there are at any time for a given connection weight, and therefore what the chance is of generating good connection weights initially. The network size determines how many degrees of freedom need to be fixed for a solution. An XOR

network consists of 4 nodes and 16 genes, whereas a Parity network consists of 21 nodes and 105 genes. Thus it is considerably more complex.

Bartlett & Downs report an average of 930 generations to achieve convergence for a population size of 50, however, they report selecting a network which solved the task within 1350 generations every run. It should be noted that the same differences apply in these results as in those described for the XOR task.

When Matyas' random optimisation technique converges, it does so after an average of 1500 presentations, yet it also fails to converge 4 out of 10 times within 5000 presentations [5].

Backpropagation requires in the order of 10000 presentations [4], which highlights one of the advantages of genetic algorithms over traditional training methods. They scale well, i.e. their performance is not affected by the size of the network as much as popular training algorithms are.

4.6 Parameter Optimisation

Considering the performance of the algorithm on these two tasks versus Bartlett & Downs, i.e. better performance on the XOR task but worse on the parity task (in terms of the number of runs which converged), it was decided to investigate the effects of parameter variation on the performance of the algorithm with the aim of tuning the algorithm.

The user definable parameters in the algorithm are

- population size
- network size
- connectivity
- stability threshold and improvement requirement
- mutation variance
- mutation increment

Variables beyond user control but subject to question are

- neuron transfer function - type, and whether it should be static or subject to compression, expansion and offset changes
- connection weights - range and distribution
- restrictions on interconnections - perhaps an exponential decrease in interlayer connectivity as the distance in layers increases is more feasible. Such a structure would be more easily built with current VLSI technology.
- selection of gene
- how to apply mutation
- whether the crossover operator is appropriate
- the way the genetic algorithm works - should it use exponential selection of parents to breed children, rather than breeding children for all couples in the population

Clearly, there are many factors which could be investigated. The investigation has focussed on the user definable parameters, with extensive testing for variations in mutation variance, mutation increment, and connectivity. Consideration is given to the variables not examined experimentally in Section 5.

Variations in population size were not investigated because its effects are assumed to be predictable. The more networks that are generated to begin with, the more diverse the gene pool, and therefore the greater the likelihood of finding a set of genes which may be combined without mutation into a network which can perform the task. The likelihood of generating a network which can do the job in the initial population is also increased. If, however, the original population does not lead to a solution by recombination, the advantages of a large population size are lost, since the expected improvement due to mutation is a function of the number of generations over which mutation is applied, rather than the population size [3].

Clearly, the performance of the algorithm is also affected by the size of the networks used. If the networks contain insufficient neurons, the algorithm will never produce a network which can perform the task. If the size of the networks is much greater than necessary, the

algorithm should be able to select a network which can perform the task, but it will not necessarily be minimal since superfluous nodes and connections are likely to be included in the final design. The algorithm will also need to do more work to eliminate those regions of the solution space which do not contribute to the task. The likelihood of converging to non-optimal solutions in the short-term is also increased since the number of non-optimal solutions increases with increased network complexity [3]. For the purposes of this investigation, the choice of 4 nodes for the XOR task is demonstrably adequate for the task.

Variations in stability threshold and improvement requirement have not yet been investigated.

Although it is unlikely that variations in mutation variance and mutation increment are independent, they have been tested independently.

The investigation was carried out using the XOR task as a standard test with the effect on the algorithm being examined for both binary and real valued outputs, i.e. for both bit error and mean square error raw scoring methods. For each choice of parameter, the program was run for a population size of 20, an improvement requirement of 0.001, and a stagnation threshold of 0.001 with a base mutation fraction of 0.0. 10 runs were performed for each parameter using 10 different random number seeds as before.

4.6.1 Binary Results

Table 4.6 contains the results for the bit-error scoring method.

Each line corresponds to a different choice of parameters. The first set of results relate to various selections for the mutation variance, σ^2. The next set relate to various choices for the mutation increment, μ_{inc}, listed here in terms of the increment in the number of genes to be mutated. The last set relates to various choices for the probability of connection or connectivity, ρ_c, listed here in terms of the fraction of the whole network which was to be connected. The parameters labelled with a star, *, represent the reference parameters used. That is, in any set of results, the parameters not being varied were held fixed at a mutation variance of 0.01, a mutation increment of 0.96 genes, and a connectivity of 0.5.

Parameter	Non-Converging Fraction	Convergence Score	Final Score	Perfection Score	Performance Index
$\sigma^2 = 0.0025$	0.7	0.70720	0.17500	0.89770	0.61998
$\sigma^2 = 0.01$ *	0.6	0.61000	0.15000	0.63575	0.49894
$\sigma^2 = 0.04$	0.5	0.56320	0.12501	0.64400	0.45805
$\sigma^2 = 0.06$	0.1	0.37065	0.02515	0.48480	0.24515
$\sigma^2 = 0.09$	0.0	0.25275	0.00033	0.43445	0.17188
$\sigma^2 = 0.12$	0.0	0.09395	0.00026	0.71545	0.20241
$\sigma^2 = 0.15$	0.0	0.11270	0.00015	0.49575	0.15215
$\mu_{inc} = 0.48$	0.6	0.63470	0.15001	0.78695	0.54292
$\mu_{inc} = 0.67$	0.6	0.62840	0.15001	0.68630	0.51618
$\mu_{inc} = 0.96$ *	0.6	0.61000	0.15000	0.63575	0.49894
$\mu_{inc} = 1.33$	0.5	0.52120	0.12502	0.68895	0.45879
$\mu_{inc} = 1.92$	0.6	0.62935	0.15000	0.69340	0.51819
$\mu_{inc} = 2.66$	0.6	0.62630	0.15008	0.78955	0.54148
$\mu_{inc} = 4.00$	0.7	0.70680	0.17508	0.79785	0.59491
$\rho_c = 0.3$	0.6	0.69520	0.15019	0.90480	0.58755
$\rho_c = 0.4$	0.8	0.84705	0.20003	0.85465	0.67543
$\rho_c = 0.5$ *	0.6	0.61000	0.15000	0.63575	0.49894
$\rho_c = 0.6$	0.3	0.42815	0.07504	0.50630	0.32737
$\rho_c = 0.7$	0.3	0.33990	0.07501	0.48930	0.30105
$\rho_c = 0.8$	0.3	0.39265	0.07501	0.72060	0.37206
$\rho_c = 0.9$	0.4	0.40250	0.10000	0.50125	0.35094
$\rho_c = 0.7, \sigma^2 = 0.09$	0.0	0.04890	0.00004	0.39335	0.11057

* = reference

Table 4.6 - Parameter Optimisation for XOR Task Bit Error Scoring Method

The columns represent different normalised performance indicators ranging between 0 and 1, with lower numbers indicating better performance. The definition of the columns is as follows:

- The Non-Converging Fraction is defined as the number of non-converging runs divided by the number of runs in the test (set of 10 runs for a parameter value).

- The Convergence Score is defined as the sum of the generation numbers at which convergence occurred for all runs in the test, divided by 20000, which represents the maximum number of generations per run, times the number of runs in the test. For non-converging runs, the generation at which convergence occurred was taken to be 2000 (the number of generations in a run). Thus, if no runs in a test had converged, the Convergence Score would be 1.0.

- The Final Score is defined as the sum of the final scores for all runs, divided by the number of runs in the test. For tests where all networks converged, the final scores are a measure of the stability in the networks at the end of the run.

- The Perfection Score is defined as the sum of the number of generations required to achieve a perfect score (attain stability as well as perform the task correctly) divided by 20000. For networks which did not converge, the number of generations required to achieve perfection was taken to be 2000, as for the Convergence Score.

- The Performance Index is defined as the average of the above four scores.

In this investigation there have been some surprising results.

Normally it is recommended to keep the amount of mutation variance in a genetic algorithm small (in the order of 0.01). In these results, it is interesting to note that the performance of the algorithm improved with increasing variance.

It was expected that the mutation increment would have little effect on performance, because it matters little for any given variance how many more genes are mutated at each generation. With the present definition of mutation increment, the amount of the gene string subjected to mutation is proportional to the length of the gene string. The size of an XOR net is 16 genes, so whether the mutation increment is 2 genes rather than 1 merely means the whole gene string is being mutated within 8 generations of stability occurring rather than 16. Over 2000 generations, this difference should have little effect, since it typically takes many more generations before mutation results in a useful variation. Nevertheless, it can be seen that

there is room for improvement by optimising the mutation increment, indicating that performance might be optimised for a particular mutation fraction. The sample size is small, however, so more runs would be required to establish a correlation.

The optimisation of connectivity is dependent on the size of the network selected and the complexity of the task to be performed. An optimal feedforward perceptron network solution to the XOR task consists of 3 nodes and 6 forward connections. For a 2 layer 2 input network as used in this case, there are 4 nodes and 24 possible connections neglecting bias connections, which are independently specified in any case. The results indicate that between 60 and 80% connectivity is optimal for the genetic algorithm to operate on this task. This may be interpreted to mean the designs produced are inefficient, since for optimum performance of the algorithm between 14 and 19 connections are used where only 6 are needed. Alternatively, the result may be interpreted to mean that for a population size of 20, between 12 and 16 random connections need to be generated at each location for the algorithm to achieve optimum performance.

The performance effects of connectivity and mutation variance are not independent. This can be seen in the bottom line of Table 4.6, in which the connectivity was set to 70%, a mutation variance of 0.09 was selected, and the mutation increment was left at 0.96 genes. The performance for this test was considerably better than the independent performance for tests using a connectivity of 70% or a mutation variance of 0.09. The results for this test appear in Table 4.7 where they are presented with the results obtained in Table 4.3 for comparison. The average generation at which convergence occurred in this test was 98.

The Parity task was also rerun with the optimal parameter values for the XOR task. The results obtained appear in Table 4.8 together with the results previously obtained. For those runs which converged, the average generation at which convergence occurred was 155, which is considerably better than before. However, 3 out of the 10 runs still failed to converge within 2000 generations.

Test Random Seed	$\rho_c = 0.4$ $\sigma^2 = 0.2$ $\mu_{inc} = 1.6$ Convergence Generation	$\rho_c = 0.7$ $\sigma^2 = 0.09$ $\mu_{inc} = 0.96$ Convergence Generation
1	129	17
2	508	161
3	4	3
4	5	164
5	329	179
6	14	3
7	56	407
8	27	24
9	30	5
10	103	15

Table 4.7 - Comparison of unoptimised and optimised XOR results

Test Random Seed	$\rho_c = 0.5$ $\sigma^2 = 0.01$ $\mu_{inc} = 4.2$ Convergence Generation	$\rho_c = 0.7$ $\sigma^2 = 0.09$ $\mu_{inc} = 1.0$ Convergence Generation
1	>2000	8
2	>2000	13
3	>2000	1000
4	>2000	31
5	982	>2000
6	467	9
7	7	>2000
8	549	8
9	479	9
10	370	>2000

Table 4.8 - Comparison of Parity results using unoptimised parameters and parameters optimised for the XOR task.

It is important to note that the definition of the mutation increment, μ_{inc}, in Tables 4.7 and 4.8 is in terms of the increment in the number of genes mutated (consistent with the other results in this section), not in terms of the incremental fraction of the gene string mutated, as in the earlier sections, where the results in the central columns of these tables were first discussed.

4.6.2 Real Results

Table 4.9 lists the results obtained for the parameter tests carried out in 4.6.1 using a mean square error scoring method instead of a bit-error scoring method.

It is again noticeable that performance improves with increasing mutation variance. A connectivity of between 60% and 80% is also desirable, however, the performance is optimised with low range mutation increments rather than mid-range increments. It should also be noticed that the combination of relatively good parameter settings achieved only a marginal improvement in performance.

Convergence occurs more often using the mean square error scoring method because the algorithm gets better feedback from the testing procedure. This is because it is able to differentiate networks better on the basis of fitness. Using the bit-error method, a network having an output of 0.7, say, when an output of 1.0 is expected, is assessed as performing as well as a network having an output of 1.0 when the respective outputs are hard limited. If, in addition, the network outputting 0.7 is generated first, and is therefore inserted into the list of ranked children first, the second network may be discarded when the quota of survivors is attained, because it is not deemed any better. On the other hand, any testing method based on real outputs will resolve the correct rank of networks on the basis of output performance.

This difference does not account for the fact that perfection occurred less frequently using the mean square error testing method, nor for the fact that the results are less easily differentiated. In fact, the mean square error scoring method resulted in extremely long periods of stability, often with no further improvement after 500 generations. Given that the initial populations generated for the two testing methods was the same and that mutations generated for the two methods would have been of the same form, what could have caused such differing results?

Parameter	Non-Converging Fraction	Convergence Score	Final Score	Perfection Score	Performance Index
$\sigma^2 = 0.0025$	0.2	0.21570	0.11736	0.60005	0.28328
$\sigma^2 = 0.01$ *	0.2	0.21580	0.11018	0.78420	0.32755
$\sigma^2 = 0.04$	0.2	0.21565	0.12159	0.78435	0.33040
$\sigma^2 = 0.06$	0.2	0.21575	0.11598	0.78425	0.32899
$\sigma^2 = 0.09$	0.1	0.12665	0.10858	0.77395	0.27729
$\sigma^2 = 0.12$	0.1	0.12035	0.09231	0.78020	0.27321
$\sigma^2 = 0.15$	0.1	0.11805	0.08908	0.68965	0.24920
$\mu_{inc} = 0.48$	0.2	0.20345	0.10762	0.61250	0.28089
$\mu_{inc} = 0.67$	0.2	0.20745	0.12792	0.61040	0.28644
$\mu_{inc} = 0.96$ *	0.2	0.21580	0.11018	0.78420	0.32755
$\mu_{inc} = 1.33$	0.3	0.30070	0.17035	0.59990	0.34274
$\mu_{inc} = 1.92$	0.3	0.30070	0.13735	0.59990	0.33449
$\mu_{inc} = 2.66$	0.3	0.30860	0.14359	0.69140	0.36090
$\mu_{inc} = 4.00$	0.3	0.30100	0.15338	0.69900	0.36335
$\rho_c = 0.3$	0.2	0.31415	0.23019	0.88585	0.40755
$\rho_c = 0.4$	0.0	0.00135	0.15652	0.89865	0.26413
$\rho_c = 0.5$ *	0.2	0.21580	0.11018	0.78420	0.32755
$\rho_c = 0.6$	0.0	0.00075	0.07581	0.89970	0.24407
$\rho_c = 0.7$	0.0	0.00065	0.07789	0.79990	0.21961
$\rho_c = 0.8$	0.1	0.17605	0.03969	0.62445	0.23505
$\rho_c = 0.9$	0.1	0.10525	0.09395	0.79525	0.27361
$\rho_c = 0.7, \sigma^2 = 0.09$	0.0	0.00065	0.04976	0.79995	0.21259

* = reference

Table 4.9 - Parameter Optimisation for XOR Task Mean Square Error Scoring Method

If the mean square error results are viewed in isolation, one notices that performance is optimised for low mutation increments. One conclusion which may be drawn from this observation is that the difficulty the algorithm has in perfecting the networks is due to too much mutation. The fact that the results are not easily differentiated may just be repeated observations of the same phenomenon.

To substantiate this claim, consider the fact that XOR nets consist of 16 genes, so the whole gene string would have been mutated after 16 generations of stability in all tests but those investigating the effects of varying mutation increment. This means that after 16 generations

of stability, all connections in the networks would have been subjected to mutation. In effect, the mutants could be viewed as new random networks, since all connections of mutants would have varied by some normally distributed number from the connections of their parents. It is of interest that for the low mutation variance of 0.0025, the perfection score was reduced, indicating that solutions either converged to perfection, or convergence occurred at more advanced stages in the runs. However, since the convergence scores for this test and the test using a mutation variance of 0.01 are roughly the same, and the same number of runs converged in these two tests, the reduced perfection score in the case of lower mutation variance can only be attributed to the selection of a greater number of stable networks, or the selection of networks which were more stable when the runs finished. Upon further investigation, it was discovered that more stable networks were selected for the low variance case as can be seen in Table 4.10, which lists the main events in the runs for the tests using a mutation variance of 0.0025 and a mutation variance of 0.01. In this Table, the Generation of Convergence is the generation at which the final score fell below 0.25 (the value expected for a single bit error for all input cases for the XOR task), the Generation of Perfection is the generation at which the final score dropped to 0.0, the Final Scores are the scores of the best network when the program terminated, and the Generation of Last Improvement is the generation at which the last improvement in the best network occurred. When a particular event was not achieved by the end of the run, the generation number was set to 2000 (the length of a run). As can be seen from the Table, a mutation variance of 0.0025 led to perfection in two instances, and did so within 4 and 15 generations of convergence respectively, whereas the higher variance did not lead to perfection at all. The conclusion is that lower variance leads to mutants which are "less random" or "closer" to their parents in the sense that the connection weights differ less, and that as optimality is approached, the likelihood of achieving optimality through the appropriate small variation in connection weights is increased.

Random Number Seed	Generation of Convergence		Generation of Perfection		Final Scores		Generation of Last Improvement	
	0.0025	0.01	0.0025	0.01	0.0025	0.01	0.0025	0.01
1	8	10	2000	2000	0.000394	0.000393	20	166
2	2	2	2000	2000	0.093031	0.026232	24	24
3	2	2	2000	2000	0.164783	0.164271	12	14
4	2000	2000	2000	2000	0.250000	0.250000	1	1
5	1	1	16	2000	0.000000	0.000001	16	12
6	2	2	2000	2000	0.206217	0.194334	7	1822
7	295	295	299	2000	0.000000	0.001269	299	1789
8	2	2	2000	2000	0.106643	0.113132	23	13
9	2000	2000	2000	2000	0.250015	0.250015	2	2
10	2	2	2000	2000	0.102549	0.102197	16	13

Table 4.10 - Events in low Mutation Variance Tests

For a population size of 20, if there was no improvement after 500 generations, then 30000 random networks would have been generated without one being found that was better. In other words, 30000 attempts were made at generating an improved set of 28 connection weights without success! It is hard to imagine that success could not have been achieved, without supposing that improvements to some connection weights were in all cases cancelled by deterioration in others. This is not an encouraging thought, and it questions the feasibility of applying the algorithm with mean square error scoring methods to binary combinatorial tasks. Kauffman & Levin indicate that as the complexity of a system increases, with a fixed mutation rate, selection becomes unable to pull an adapting population to those local optima which may be achieved through mutation via fitter variants [3]. In this way, any problems experienced with a simple XOR network would become more noticeable on more complex tasks, or rather, with more complex networks.

The bit-error scoring method may not have been troubled as much by the effects of competitive connection weight improvements, because the hard limiting of the outputs eliminated differences in the outputs between unstable well performing networks and more stable poorer performers. The bit-error method therefore only needs to roughly optimise outputs before concentrating on obtaining stable networks.

5 Further investigation

5.1 Verification of Results

The results indicate certain trends for parameter optimisation on the XOR task which led to an improvement in the Parity task as well. The question which this raises, and remains unanswered, is can these results be generalised? Which other tasks can they be applied to? To answer this question definitively would require carrying out tests and examining the results for other tasks. The XOR task is an example of a Boolean task. There is evidence to suggest that the trends observed and the problems encountered in this investigation occur in any complex optimisation problem (of which the selection of neural networks to perform Boolean tasks is a special case) [3]. It is likely, therefore, that the trends would be observed in general, however, more tests need to be carried out to verify this.

5.2 Extensions to results

5.2.1 Rate of Mutation Application

The severity of mutation and the granularity of mutation have been investigated in the guise of the mutation variance and the mutation increment. The factors controlling the rate of mutation application, stagnation threshold and improvement requirement have not been investigated. For the sake of completeness, this should be done.

5.2.2 Changing Environment

The results obtained relate to a static environment, or one in which the task does not change during a run. Thus, testing has focussed on the ability of a network to perform a task, rather than the ability of a network to learn a task, which is a desirable trait for networks which must adapt to a changing environment. Adaptation to a changing environment should be investigated as a generalisation of the work carried out.

5.2.3 Identification of Suitable Substructures

It is still unknown which substructures, if any, impart high fitness. The performance of the algorithm could be improved if it could decide which particular parts of networks need to be combined to produce a high performer. Testing of networks is performed holistically, i.e. the network is treated as a black box as far as the testing algorithm is concerned, and its performance at all parts of the task is examined in total. If networks were evaluated with different permutations of genes, then the genetic algorithm might be modified to understand which substructures are worth keeping, and which substructures do nothing to increase fitness. By trying different combinations of genes, the algorithm may be able to identify those structures which are good. To apply such an exhaustive search would, in general, be impractical. To this end the selection and evaluation of genes could be done randomly.

5.3 Variations to the Model

5.3.1 Breeding Model

The loop used in this investigation

```
REPEAT FOREVER
        {
                CHILDREN = BREED(ALL COUPLES FROM SET OF PARENTS)
                SCORES(CHILDREN+PARENTS) =
                        TEST_AGAINST_ENVIRONMENT(CHILDREN+PARENTS)
                PARENTS = SELECT_BEST(CHILDREN+PARENTS) BASED ON
                        SCORES(CHILDREN+PARENTS)
        }
```

is often replaced in the literature by the loop

```
REPEAT FOREVER
        {
                PARENTS = CHOOSE_RANDOM_COUPLE(POPULATION)
                CHILDREN = BREED(PARENTS)
                FOR EACH CHILD
                        IF SCORE(CHILD) > SCORE(WORST(POPULATION))
                                DISCARD WORST(POPULATION)
                                ADD CHILD TO POPULATION
                        ELSE
                                DISCARD CHILD
        }
```

The first of these loops only allows progress via fitter variants, whereas the second allows for progress by incorporating features of less fit variants and is therefore more powerful.

The difference may be subtle, yet it can defer specialisation, a problem which occurs with the first loop, when all networks in the population have converged to have almost identical structures and connection weights. Because the networks are virtually identical, they tend to produce similar offspring, which in turn perform roughly the same. The ability to distinguish between individuals is therefore lost, and the ability to improve the population is restricted to the capabilities of the mutation operator. Thus, the algorithm becomes purely mutation driven, rather than primarily recombination driven, and the genetic algorithm performs more like a random search algorithm. Another problem is that the genetic heritage of the networks is lost because ancestors are discarded when an improved network swamps the population, which can happen when all children spawned from the improved network are improved performers destined to replace all other members of the population. Thus recombination with ancestors, to incorporate genes which were previously poorly regarded because they had not realised their full potential is not possible and ancestral genes which may only now prove useful can only be realised or reappear by mutation. For example, when an improved individual is produced for the XOR task, the entire population is replaced by it's children within three generations (repetitions of the loop). From this it can be estimated that the earliest possible ancestors an improved network could combine with are contemporaries of its great-grandparents.

The second loop suffers less from this problem because ancestors are kept until they are the worst performers in the population and a child with higher fitness is produced. The criterium for selecting a couple of parents is on the basis of fitness (and in some implementations, on the fitness of their offspring). The population is ranked on this criterium and a couple is chosen from the list of ranked individuals with exponentially decreasing likelihood towards the low fitness end of the list. Breeding therefore, does not occur "simultaneously" for all possible couples in the population as in the first model, but for a selected couple at a time.

One problem with this method, as with the first, is that a lot of identical networks can be generated, especially when the population converges towards a local optimum. This can be avoided by searching the population for networks which differ by less than some tolerance from a newly generated individual. If a duplicate exists, then the child would not be added. If the population is ranked on fitness, then only those networks with relatively similar fitness need to be checked. However, the time to compare the connections of two networks is proportional to the size of the network squared, so for large populations of large individuals this approach may be too computationally demanding. Another solution might be to eliminate individuals on the basis of fitness. If the spread of scores is large enough to distinguish networks on this basis, then by defining uniformly sized fitness intervals over the range of scores, all individuals but the best in each interval could be eliminated. The remaining networks all score differently, so the population could be assumed to have regained acceptable diversity.

The Genetic Retention Law states that the proportion of a gene in a population must remain constant during breeding. Bentink comments [1, p.24] that the elimination of duplicates contravenes the Genetic Retention Law, and hence it should not be adopted. This is incorrect. The Genetic Retention Law ensures the process of breeding is not biased towards any particular gene, and hence that particular genetic structures are not favoured by the breeding process. The elimination of duplicates occurs after breeding, when the Genetic Retention Law no longer applies. Therefore it cannot be contravening the law. The purpose of obeying the Genetic Retention Law is to avoid stagnation brought about by the production of overwhelming numbers of (almost) identical individuals. The elimination of duplicates has similar aims and should therefore be viewed as a complementary concept applying to individuals at a different stage in their development.

Whether the Genetic Retention Law is a valid concept in nature is questionable. For any gene or characteristic which partitions a population into 2 groups (those which have the gene, and those which don't), the ratios of these 2 groups can only be maintained by the production of offspring in the same proportions. Since any normal child cannot possess a fraction of a gene, multiple children must be bred simultaneously in the required proportions to obey the law strictly.

5.3.2 Network Model

The network model used by this implementation is that of a completely connected graph, where each connection is a two way independently variable line. This structure is impossible to implement using current VLSI technology because of the potentially large number of overlapping wires. It is also unlikely that semiconductor technology would ever advance to the stage where processors would be capable of having fan ins and fan outs in the order of 1000, which is the case with neurons in the human brain. Whether in fact this would be necessary given that artificial neurons switch much faster than natural neurons is another matter. However, the fact remains that the model represented by this implementation becomes impossible to implement with more than 4 neurons, given that two layers of metallisation is the current state of VLSI art. Some restrictions on internodal connections is required to implement the designs developed by the algorithm.

To implement natural neural structures would ideally require the ability to implement three-dimensional tree structures. Current technology is restricted to planar implementations in which the tree is a less than ideal computational model because of the differences in wire lengths at different levels of the tree. In a three dimensional tree, however, connection lengths could be independent of depth if it were implemented as a sphere with the root at the centre. However, using this hypothetical structure, unlimited processor interconnections are also not feasible. A restriction which seems reasonable to impose is to enforce variation in the density of interconnections between layers. The further a layer is from a particular node (in terms of intervening layers), the less likely it would be that the node has any connections with that

layer. The distribution of connections between layers might be normal or exponentially decreasing with increasing distance.

A second variation worth investigating is to examine the fitness of networks whose *neural functions* vary, rather than whose *connection weights* are varied during training. Having decided on a particular base neural function, variations might be to compress or expand the function; offsets and saturation levels should also be allowed to vary. This would seem to be a more accurate neuronal model and may prove to result in more efficient networks and training algorithms since optimising the connections weights is a dimensionally bigger problem.

5.3.3 A Network Farm?

The genetic algorithm, as described here, is a method by which good substructures of networks are recombined to form new networks having the same gross structure as their parents. This gross network structure is defined by the network designer, and may be based on assumptions which are invalid, or constrain the algorithm to perform poorly. A variation could be envisaged where the population of networks which represent approximate solutions to a task is replaced by a pool of networks having differing structures and capable of performing different tasks. Providing the genetic structure of any network in the pool can be represented, with appropriate combination operators, networks could be combined much like lego blocks to form new structures. Similarly, networks might be blown apart to form smaller structures so that there is always an adequate supply of fundamental building blocks. The pool could contain fundamental network blocks such as MAXNET [7], all networks which perform the two input Boolean functions, and good solutions to previous tasks. When a task is to be performed by a network, the algorithm searches for networks in the pool which can perform the task. If none are available, then they are built up from the blocks available, or randomly generated. The connections to the inputs are generated and the genetic algorithm proceeds.

6 Conclusion

The performance of a genetic algorithm applied to the selection of neural nets has been investigated in this project.

The algorithm was tested on the XOR task using various mutation strategies. Of the mutation strategies investigated, a strategy which allowed the user to specify the severity of a point mutation, the rate at which mutation was applied to a network, as well as the granularity of mutation application performed best and was thought to be most flexible. A variation in which the severity of point mutations, or the mutation variance is itself allowed to vary is thought to be an even better approach but it was not implemented. Results obtained with the best strategy on two Boolean tasks indicate the genetic algorithm compares favourably with the popular backpropagation training method.

With the aim of optimising the performance of the algorithm, the performance was investigated for several user definable algorithmic parameters and two different scoring methods. The observation was made that the performance of the algorithm is governed by the choices made for these parameters, provided the network structure chosen is appropriate for the task. It was also observed that the choice of scoring method influences the effect of these parameters on the performance of the algorithm.

Tests carried out on the XOR task indicate that:

- When the networks are tested using a mean square error scoring method, the mutation variance should be at least as low as 0.0025 if perfection is desired. If it is more desirable to achieve rapid convergence, a mutation variance as high as 0.15 would be more appropriate. If a bit-error scoring method is used, as seems reasonable for Boolean tasks, variances as high as 0.15 provide good convergence and perfection.

- There is evidence to suggest that for the bit-error scoring method the granularity of mutation application, or the increment in the amount of the network mutated with prolonged stagnation, can be optimised. When the mean square error scoring method

is used, the mutation increment should be kept low.

- The rate at which mutation is applied to the networks needs to be investigated. It may be that with lower rates of mutation application, the mean square error method would achieve perfection more easily.

- The connectivity of the original population seems to be a factor affecting performance. For the XOR task, 60 to 80% connectivity seems optimal for both scoring methods. This figure indicates the final networks contain more connections than are needed, or that 12-16 random connections need to be generated at each possible location to achieve optimal performance.

It was unexpected that the ability of the algorithm to perfect a network would be lower for the mean square error testing method than for the bit-error scoring method because it was felt the genetic algorithm would get better feedback about the performance of the networks from the testing procedure. The poor performance of the algorithm on this score is thought to be due to the cancellation of beneficial changes to some connection weights by detrimental changes to others. It is felt that this problem would become more pronounced as the network size grows. Nevertheless, the bit-error scoring method performed reasonably well on this score, thus for tasks where it is possible to hard-limit the output, this method should be adopted in preference.

Whether genetic algorithms or training algorithms are better at providing neural networks remains to be seen. In this investigation it was found that the algorithm can be difficult to apply to the "provision problem" without a theoretical basis for making appropriate choices of algorithmic parameters.

Appendix A Genetic Algorithm Concepts

A genetic algorithm is a probabilistic optimisation procedure loosely based on the processes of reproduction, natural selection and mutation. Before describing the algorithm, some terms used in the description need to be defined.

To use a genetic algorithm we need to be able to describe solutions to a problem as a linear list of solution characteristics. Each characteristic is called a gene. Candidate solutions are called individuals and are described by a list of genes called a gene string or chromosome. A chromosome may be thought of as being a representation of the genotype of the individual and the solution it expresses as representing the phenotype of the individual. Each individual has an attribute called its fitness which is a measure of how well it solves the problem. The fitness associated with an individual is usually some function of the difference between the solution it expresses and the expected solution.

This is how the algorithm works:

Given a set of suboptimal or incomplete solutions to a problem, the 2 best elements on the basis of fitness are chosen to become parents of the next generation. Children are formed by splitting the chromosomes of the parents at some random position and recombining the complementary fragments. There is a good chance that the child will have even greater fitness. The algorithm proceeds by repeatedly selecting suitable parents and producing offspring until an optimal child is found.

These concepts are demonstrated by the following example [11]:

VECTOR is a game played by two people. Player A secretly writes down a string of 6 binary digits. Player B must deduce the string using as few guesses as possible.

This demonstration will show that it is possible to play the part of Player B using a genetic algorithm. With reference to Figure A.1, Player B starts by generating 4 random strings which player A scores on the basis of matching bits. Player B next selects the 2 highest scoring

strings and generates 4 new guesses by splitting the previous best solutions randomly and recombining them. In this case, Player B chooses to split the solutions after the second and fourth most significant bits in successive generative steps. The resulting strings are again scored by Player A.

Parent String		Guess/Child String		Score
		A)	010101	1
		B)	111101	1
		C)	011011	4
		D)	101100	3
C)	01:1011	E)	01:1100	3
D)	10:1100	F)	10:1011	4
C)	0110:11	G)	0110:00	4
D)	1011:00	H)	1011:11	3
F)	1:01011	I)	1:11000	3
G)	0:11000	J)	0:01011	5
F)	101:011	K)	101:000	4
G)	011:000	L)	011:011	4
J)	0010:11	M)	0010:00	5
K)	1010:00	N)	1010:11	4
J)	00101:1	O)	00101:0	6
K)	10100:0	P)	10100:1	3

Figure A.1 - VECTOR Example

Picking 2 of the top scorers, F and G say, Player B generates 4 more children by splitting after the first and third most significant bits. Choosing J and K as the parents of the next generation, again 4 children are generated and this time a perfect score results. The advantage over a random search is obvious, since only 16 guesses were made as opposed to the 32 expected for the 64 possible strings.

To see how the genetic algorithm optimises the solution characteristics the strings F and G

should be considered. These strings are points on the vertices of a unit hypercube in 6 dimensional space. Any point can be considered as a sample of the points in a lower dimensional hyperplane in this space, so that the fitness of a point is an estimate of the average fitness of all points in that hyperplane. For example, if * is a wildcard character, denoting either a 0 or a 1, then F is a sample of the points on the two-dimensional hyperplane H_1 of Figure A.2 and of the points on the two-dimensional hyperplane H_2, and F's fitness is an estimate of the average fitness of these hyperplanes [5].

H_1)	1010**	H_3)	*01011
H_2)	*0101*	H_4)	0*****

Figure A.2 - Hyperplanes in 6 dimensional space

Given that F and G have been chosen as two parents because they have high fitness, a crossover might occur after the most significant bit. Since F's fitness is an estimate of the fitness of points on the one-dimensional hyperplane H_3 and G's fitness is an estimate of the fitness of points on the five-dimensional hyperplane H_4 the offspring produced, J, is the point at the intersection of these two (high-fitness) hyperplanes, and it can be expected that it will be a high-fitness individual.

Each individual is a sample of 2^6 hyperplanes, but many of these hyperplanes will be destroyed by the crossover operation. However, for a population of N individuals, the number of hyperplanes which are usefully processed is N^3 [12]. Holland calls this implicit parallelism - a genetic algorithm processing a small number of bit strings automatically processes a large number of hyperplanes. Because bit strings are removed on the basis of fitness, a genetic algorithm reduces the effective dimensionality of the optimisation problem. For example, if an individual always exhibits poor performance when a particular bit has a value 1, the population will quickly have few individuals with a 1 in that position; the dimensionality of the problem is reduced. As the algorithm progresses, it can be expected that poor-performing hyperplanes will become less common in the population [5]. Once bad parent strings are

eliminated their offspring are automatically eliminated. Each step is evaluating present and potential strings in parallel.

This example should demonstrate a problem with the genetic algorithm as stated. That is if no string contained a 1 in the third most significant position when initially generating the random guesses, then no amount of selection and recombination could have generated the secret code. To this end the concept of mutation is introduced into the algorithm when no further improvement is possible and the solution is known to be suboptimal. For example children might be formed with a finite probability that one or more bits are incorrectly copied, thereby potentially avoiding convergence to a suboptimal solution.

Appendix B Program Listings

B.1 System Overview

The main functions and interactions of files presented in the listings is as indicated in Figure B.1. In this figure, boxes represent modules, with the direction of arcs connecting modules indicating the hierarchy of interaction. The purpose for interaction is indicated along the arc.

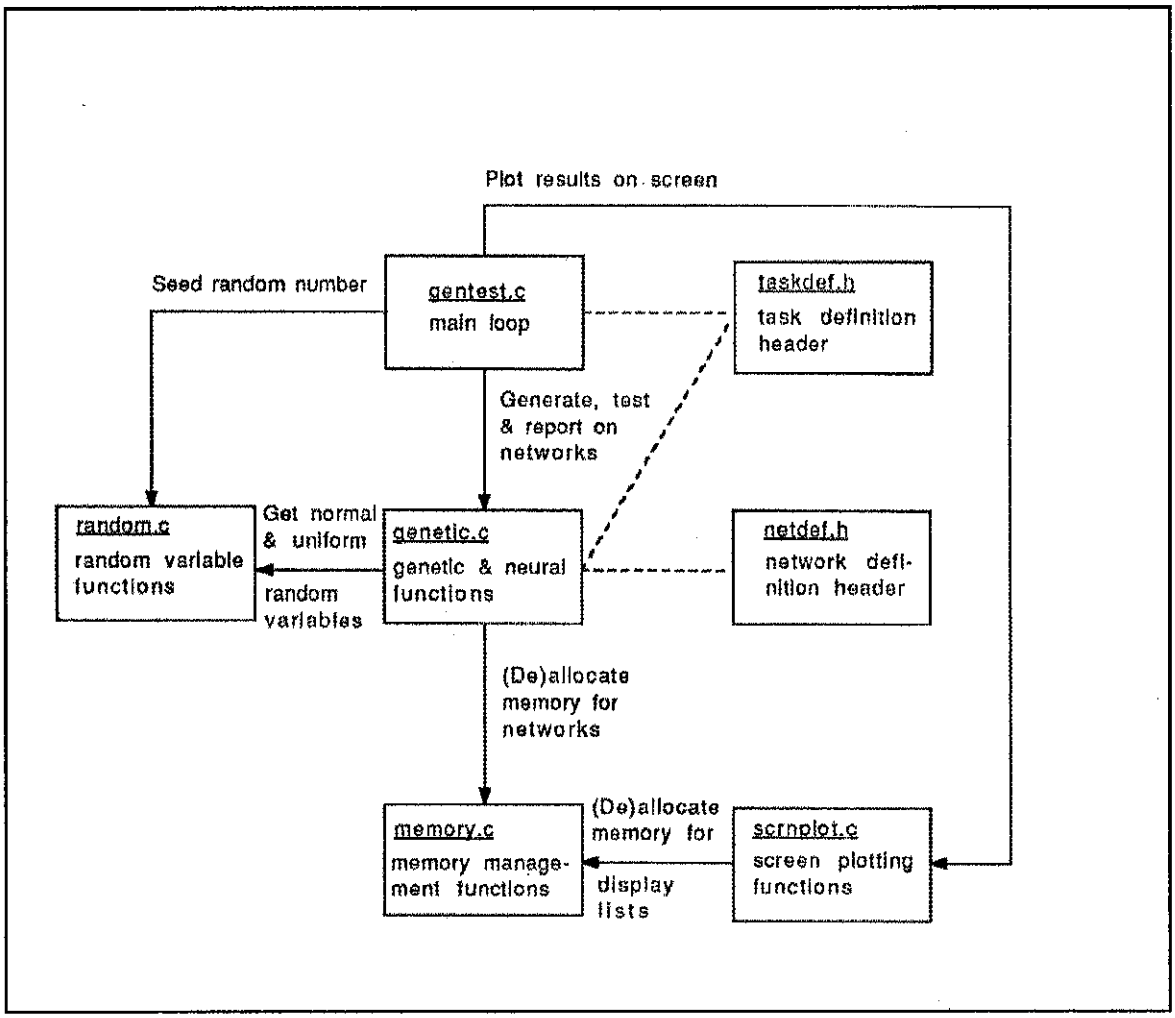


Figure B.1 - System Overview

B.2 Program Files

```
/*****************************************************************
**                                                              **
**   filename:      gentest.c                                   **
**   programmer:    m.bentink                                   **
**   modified:      901015ofd                                   **
**   description:   main routine for genetic selection of neural **
**                  networks                                    **
**                                                              **
*****************************************************************/

#include  <stdio.h>
#include  <string.h>
#include  <math.h>

/*
**   TurboC specifics - comment out for run on Apollo
*/
#include  <time.h>

/*
**   Apollo specifics - comment out for run on PC
*/
/*#include     <sys/types.h>
*/

#include  "extendc.h"          /*   C extensions */
#include  "netdef.h"           /*   Network definitions */
#include  "genetic.h"          /*   Genetic routines */
#include  "random.h"           /*   Random number functions */
#include  "scrnplot.h"         /*   Screen plot routines */

/*
**   Local definitions
*/

#define   MIN_SCORE 0.000001

/*
**   Local prototypes
*/

void      DefineFileNames(    char run_name[] );

/*
**   User defined global variables
*/

int       DETAILS;                  /* boolean indicating detailed */
                                    /* reporting is required */

float     PCON;                     /* probability of connection in */
                                    /* randomly generated nets */

float     MUTATION_INCREMENT;       /* mutation factor increment */
```

```
float     MUTATION_FACTOR;          /* factor applied to generation */
                                    /* scores to determine */
                                    /* likelihood of mutation */

float     MUTATION_FACTOR_BASE;     /* base mutation factor */

float     MUTSD;                    /* standard deviation of */
                                    /* mutation operator */

float     STAGNATION_THRESHOLD;     /* expected minimum change in */
                                    /* average score below which */
                                    /* the population is deemed to */
                                    /* have stagnated */

float     IMPROVEMENT_REQUIREMENT;  /* expected minimum change in */
                                    /* best score below which the */
                                    /* population is deemed not to */
                                    /* be improving quickly enough */

int       MAX_POPULATION;           /* maximum population size */
                                    /* before breeding */

int       MIN_POPULATION;           /* minimum population size */
                                    /* before breeding */

int       MAX_GENERATION;           /* maximum number of */
                                    /* generations in run */

float     PREVIOUS_BEST;            /* best score from previous */
                                    /* generation */

float     PREVIOUS_AVERAGE;         /* average score from previous */
                                    /* generation */

int       STABILITY_COUNT;          /* count of number of */
                                    /* successive generations with */
                                    /* no improvement */

/*
**   names of files used by program
*/

char STARTIN[30];              /*   input network file list */
char DESCOUT[30];              /*   output network descriptions */
char GENOUT[30];               /*   output generation statistics */
char SUMOUT[30];               /*   output test result summary */

/*-------------------------#########----------------------------*/
```

```
void main(     void )

/*
**   the main procedure gets the user parameters and runs the genetic
**   algorithm until finished
*/

{
     des        *PARENTS, *CHILDREN, *TEMP;
     int        N, I, NO_SURVIVORS, GRAPHS, RPTLVL;
     float      B E S T _ S C O R E ,    A V _ S C O R E ,    m u t v a r ,
                WORST_SCORE;
     unsigned   SEED;
     char       RUN_NAME[30];

     int        GENERATION_NO = 0;
     float      MUT_SD, MUT_FRACT;
     FILE       *OUT_FILE;

     printf( "Testing Genetic Algorithm on %s Task\n\n", TASK_NAME );

/*
**   the user specifies a run name which is used to derive the run log
**   file names and the input list of start networks
*/

     strcpy( RUN_NAME, "1234567890" );
     while ( strlen( RUN_NAME ) >= 9 ) {
          printf( "Enter run name (<9 characters): " );
          scanf( "%s", RUN_NAME );
     }

     DefineFileNames( RUN_NAME );

/*
**   the user is requested to specify the level of on-screen reporting
**   desired to monitor the run. According to the request, the
**   appropriate boolean variables are set
*/

     RPTLVL = -1;
     while ( RPTLVL < 0 || RPTLVL > 2 ) {
          printf( "Display details 0=None/1=Graphs/2=Full: " );
          scanf( "%d", &RPTLVL);
     }

     switch ( RPTLVL ) {

          case 0:
               GRAPHS = FALSE;
               DETAILS = FALSE;
               break;
```

```
            case 1:
                GRAPHS = TRUE;
                DETAILS = FALSE;
                break;

            case 2:
                GRAPHS = TRUE;
                DETAILS = TRUE;
                break;
        }

    /*
    **  a random number seed may be specified by the user to control
    **  whether a new sequence of random numbers is to be generated
    */

        printf( "\nEnter random number seed : " );
        scanf( "%u", &SEED );

    /*
    **  seed the random number generator
    */

        Randomize( SEED );

    /*
    **  population sizes are restricted by the amount of main memory
    **  available therefore target machine type sizes may be used in
    **  determining the maximum pre-breed population size. Memory usage
    **  is then calculated on the basis of post-breed population sizes
    */

        printf( "\nSize of a float : %d bytes\n", sizeof( float ) );
        printf( "Size of a double : %d bytes\n", sizeof( double ) );
        printf( "Size of each descriptor : %d bytes\n", sizeof( des ) );

        printf( "\nEnter population minimum : " );
        scanf( "%d", &MIN_POPULATION );

        printf( "Enter population maximum : " );
        scanf( "%d", &MAX_POPULATION );

        printf( "\nPeak working area : %ld Kbytes\n\n",
            (long) MAX_POPULATION * (long) (MAX_POPULATION + 2)
                * (long) sizeof( des ) / (long) 1024 );

        printf( "\nEnter probability of connection in random nets : " );
        scanf( "%f", &PCON );

    /*
    **  the number of generations in the run may be specified so that
    **  orderly termination can occur
    */
```

```
        printf( "\nEnter number of generations in run : " );
        scanf( "%d", &MAX_GENERATION );

    /*
    **  the mutation parameters probably affect the run the most
    **
    **  more comment needed here
    */

        printf( "\nEnter stagnation threshold : " );
        scanf( "%f", &STAGNATION_THRESHOLD );

        printf( "Enter improvement requirement : " );
        scanf( "%f", &IMPROVEMENT_REQUIREMENT );

        printf( "\nEnter base mutation factor : " );
        scanf( "%f", &MUTATION_FACTOR_BASE );

        MUTATION_FACTOR = MUTATION_FACTOR_BASE;

        printf( "Enter mutation factor increment : " );
        scanf( "%f", &MUTATION_INCREMENT );

        printf( "Enter mutation operator variance : " );
        scanf( "%f", &mutvar );
        MUTSD = sqrt( (double) mutvar );

    /*
    **  the algorithm commences by reading in any networks specified in
    **  the user-defined starting network list
    **
    **  initialize the number of networks, and
    **  read in any input networks creating a list of descriptors pointed
    **  to by PARENTS
    */

        N = 0;
        PARENTS = Read_Networks( &N );

    /*
    **  supplement deficient starting populations with randomly generated
    **  networks
    */

        for ( I=N+1 ; I<MIN_POPULATION+1 ; I++ ) {

    /*
    **  generate a random network
    */

            TEMP = Generate_Rnd_Net( I, GENERATION_NO );
```

```
/*
**    add it to the list of PARENTS
*/

          PARENTS = Append_Des( TEMP, PARENTS );
     }

/*
**    rate the nets returning best and average scores, and
**    report on their performance
*/

     Rate( PARENTS, &BEST_SCORE, &AV_SCORE );

     PARENTS = Sort( PARENTS );

     Report(    PARENTS,    BEST_SCORE,    AV_SCORE,    N,    N,
          GENERATION_NO, MUTSD, MUTATION_FACTOR );

/*
**    setup graphs if graph level of reporting selected
*/

     if    ( GRAPHS ) {
          InitializeScores();
          AddToScoreList( GENERATION_NO, BEST_SCORE, AV_SCORE );
     }

/*
**    added for inline generation and testing of children
**    determine worst score of generation 0
*/

     CHILDREN = PARENTS;

     while ( CHILDREN->NEXT != NULL )
          CHILDREN = CHILDREN->NEXT;

     WORST_SCORE = CHILDREN->SCORE;

     NO_SURVIVORS = N;

     PREVIOUS_BEST = BEST_SCORE;
     PREVIOUS_AVERAGE = AV_SCORE;
     STABILITY_COUNT = 0;
     MUT_FRACT = 0.0;
     MUT_SD = MUTSD;
```

```
/*
**    main loop
*/

      while ( GENERATION_NO<MAX_GENERATION && BEST_SCORE>MIN_SCORE )
      {

            ++GENERATION_NO;

            PARENTS = Generate( PARENTS, &N, GENERATION_NO, MUT_FRACT,
                  MUT_SD,    &BEST_SCORE,    &AV_SCORE,    &WORST_SCORE,
                  &NO_SURVIVORS);

/*
**    report on the performance of the survivors
*/

            Report( PARENTS, BEST_SCORE, AV_SCORE, NO_SURVIVORS, N,
                  GENERATION_NO, MUT_SD, MUT_FRACT );

/*
**    plot the scores on the screen if required
*/

            if    ( GRAPHS )      {
                  AddToScoreList( GENERATION_NO, BEST_SCORE, AV_SCORE );
                  PlotScores();
            }

/*
**    calculate the new mutation potential
*/

            Calc_Mutn( BEST_SCORE, AV_SCORE, GENERATION_NO, &MUT_SD,
                  &MUT_FRACT );

            if    ( GRAPHS )
                  printf( "\nMUTATION fraction = %f st.dev. = %f\n",
                        MUT_FRACT,MUT_SD);
      }

      OUT_FILE = fopen( DESCOUT, "a" );

      Write_Des( PARENTS, OUT_FILE );

      fclose( OUT_FILE );
}

/*-------------------------#########----------------------------*/
```

```
void DefineFileNames(    char RUN_NAME[] )

/*
**   defines filenames used to report results in and extract start
**   nets from
*/

{

/*
**   define input network file list name <run_name>.BGN
*/

     strcpy( STARTIN, RUN_NAME );
     strcat( STARTIN, ".bgn" );

/*
**   define network description file name <run_name>.DSC
*/

     strcpy( DESCOUT, RUN_NAME );
     strcat( DESCOUT, ".dsc" );

/*
**  define generation statistics file name <run_name>.GEN
*/

     strcpy( GENOUT, RUN_NAME );
     strcat( GENOUT, ".gen" );

/*
**   define performance summary file name <run_name>.SUM
*/

     strcpy( SUMOUT, RUN_NAME );
     strcat( SUMOUT, ".sum" );
}
/*----------------------------##########------------------------------*/
```

```
/*****************************************************************
**                                                              **
**   filename:      genetic.h                                   **
**   programmer:    o.diessel                                   **
**   modified:      901015ofd                                   **
**   description:   specifies types used by genetic routines    **
**                  contains prototypes of genetic routines     **
**                                                              **
*****************************************************************/

/*
**   Genetic definitions
*/

#define   NO_GENES  NO_NEURONS*((NO_LAYERS + 1) + 1)

/*
**   Genetic type definitions
*/

typedef   float     net_in_out[NO_INPUTS];
typedef   short     chromosome[NO_NEURONS][(NO_NEURONS+NO_INPUTS)];
typedef   float     network_state[NO_NEURONS + NO_INPUTS];
typedef   short     bit_array[NO_INPUTS];
typedef   float     weight_matrix[NO_NEURONS][(NO_NEURONS+NO_INPUTS)];
typedef   float     biasing[NO_NEURONS];
typedef   float     gene_type[NO_INPUTS];

typedef   struct    descr {

                        weight_matrix  WEIGHTS;
                        biasing        BIAS;
                        long           CREATURE_NO;
                        int            GENERATION_NO;
                        long           PARENT_A;
                        long           PARENT_B;
                        char           BREEDING_METHOD;
                        float          SCORE;
                        float          ACTIVATION;
                        float          DELTA_ACTIVATION;
                        struct descr   *NEXT;

                    } des;

typedef   struct    gene_string {

                        gene_type           GENE;
                        struct gene_string  *NEXT;

                    } gene_str;
```

```
/*
**   Genetic prototypes
*/

void       Dispose_Gene(       gene_str        *gene );

void       Dispose_Des(        des             *descriptor );

float      Calc_Mutn(          float           best_score,
                               float           average_score,
                               int             generation_number,
                               float           *mutation_std_dev,
                               float           *mutation_fraction );

des        *Gene_To_Des(       gene_str        *genes );

gene_str   *Des_To_Gene(       des             *descriptor );

void       Swap_Gene(          gene_str        *gene1,
                               gene_str        *gene2 );

void       Add_Details(        des             *child,
                               des             *parent1,
                               des             *parent2,
                               int             generation,
                               int             number,
                               char            breeding_method );

des        *Append_Des(        des             *descriptor,
                               des             *descriptor_list );

void       Write_Des(          des             *descriptor,
                               FILE            *out_file );

int        Read_Net(           weight_matrix   weights,
                               biasing         bias,
                               FILE            *in_file );

des        *Read_Networks(     int             *number );

void       Integer_To_Bits(    unsigned        integer,
                               bit_array       bits );

int        Bits_To_Integer(    bit_array       bits );

float      Neuron_Transf(      float           neuron_input);

void       Schmidt(            net_in_out      network_out_reals,
                               bit_array       net_out_bits );

void       Bits_To_Reals(      bit_array       net_in_bits,
                               net_in_out      net_in_reals);

int        No_Bit_Errors(      bit_array       net_out,
                               bit_array       expected_out);
```

```
des         *Generate_Rnd_Net(  int             number,
                                int             generation );

des         *Remove_Des(        des             *item,
                                des             *list );

des         *Sort(              des             *des_list );

void        Score(              des             *descriptor,
                                int             num_errors );

float       Activate_Network(   net_in_out      net_in,
                                net_in_out      net_out,
                                weight_matrix   weights,
                                biasing         bias,
                                float           *average_activation );

void        Test(               des             *descriptor );

void        Rate(               des             *des_list,
                                float           *best_score,
                                float           *average_score );

des         *Breed(             des             *parent1,
                                des             *parent2,
                                int             generation,
                                int             number );

des         *Mutate(            des             *parent,
                                float           mutation_fraction,
                                float           mutation_std_dev,
                                int             generation,
                                int             number );

void        Report(             des             *parents,
                                float           best_score,
                                float           av_score,
                                int             no_survivors,
                                int             no_generated,
                                int             generation,
                                float           mutn_sd,
                                float           mutn_fract );

des         *Create_Generation( des             *parents,
                                int             *number_generated,
                                int             generation,
                                float           mutation_fraction,
                                float           mutation_std_dev );

des         *Select(            des             *children,
                                int             *num_survivors,
                                float           best_score,
                                float           av_score,
                                int             generation );
```

```
void        DisplayList(          des             *descriptor_list );

void        DisplayDescriptor(    des             *descriptor );

des         *Generate(            des             *parents,
                                  int             *no_generated,
                                  int             generation_no,
                                  float           mutation_fraction,
                                  float           mutation_std_dev,
                                  float           *best_score,
                                  float           *av_score,
                                  float           *worst_score,
                                  int             *no_survivors );

int         DisplayTest(          des             *descriptor );
```

```
/*****************************************************************
**                                                              **
**   filename:      genetic.c                                   **
**   programmer:    m.bentink                                   **
**   modified:      901017ofd                                   **
**   description:   contains genetic and neural functions       **
**                                                              **
*****************************************************************/

#include  <stdio.h>
#include  <math.h>

/*
**   TurboC specifics - comment out for run on Apollo
*/
#include  <time.h>

/*
**   Apollo specifics - comment out for run on PC
*/
/*   #include  <sys/types.h>
*/

#include  "extendc.h"     /*   C extension */
#include  "netdef.h"      /*   network definition file */
#include  "genetic.h"     /*   genetic function prototypes */
#include  "random.h"      /*   random variable prototypes */
#include  "memory.h"      /*   memory allocation prototypes */
#include  "taskdef.h"     /*   network task definition file */

/*
**   constants used in the neuron transfer function:
**   OUT = AN*IN^5 + BN*IN^4 +  CN*IN^3 + DN*IN^2
*/

#define   AN    6.4
#define   BN    -16.0
#define   CN    10.8
#define   DN    -0.2

/*----------------------------#########-------------------------------*/

void Dispose_Gene(  gene_str  *GENE_LIST )

/*
**   deletes the genestring pointed to by GENE_LIST and places memory
**   allocated to it back into the free memory pool
*/

{
     gene_str  *TEMP;
```

```
/*
**    free up space for each gene in GENE_LIST
*/

      while ( GENE_LIST != NULL ) {

            TEMP = GENE_LIST;
            GENE_LIST = GENE_LIST -> NEXT;
            TEMP->NEXT = NULL;
            Free( TEMP );
      }
}

/*---------------------------##########-----------------------------*/

void Dispose_Des(   des   *DES_LIST )

/*
**    deletes the list of network descriptors pointed to by DES_LIST
**    and places memory allocated to them back into the free memory
**    pool
*/

{
      des   *TEMP;

/*
**    free up space for each descriptor in DES_LIST
*/

      while ( DES_LIST != NULL ) {

            TEMP = DES_LIST;
            DES_LIST = DES_LIST -> NEXT;
            TEMP->NEXT = NULL;
            Free( TEMP );
      }
}

/*---------------------------##########-----------------------------*/

float      Calc_Mutn(      float      BEST_SCORE,
                           float      AV_SCORE,
                           int        GENERATION_NO,
                           float      *MUT_SD,
                           float      *MUT_FRACT )

/*
**    returns a number representing the likelihood of mutation
**    (mutability) during the next generation based on the amount of
**    change in the best and average scores between this and the last
**    generation
**
**    determines the standard deviation applied to the mutation
**    operator in Mutate (MUT_SD) and the fraction of genes to mutate
```

```
**    in Mutate (MUT_FRACT)
**
**    should be called Calc_Mutability
*/

{
      extern    float     MUTATION_FACTOR;
      extern    float     MUTATION_FACTOR_BASE;
      extern    float     MUTATION_INCREMENT;

      extern    float     STAGNATION_THRESHOLD;
      extern    float     IMPROVEMENT_REQUIREMENT;
      extern    int       DETAILS;
      extern    float     PREVIOUS_BEST;
      extern    float     PREVIOUS_AVERAGE;
      extern    int       STABILITY_COUNT;
      extern    float     MUTSD;
      extern    int       MAX_GENERATION;

      float     MUTN, SD;

/*
**    report on old mutation factor
*/

      if    ( DETAILS )
            printf( "Old mutation factor = %8.6f\n", MUTATION_FACTOR );

/*
**    if the population average is stagnating, or the population best
**    is not improving, then increase the mutation factor, else reduce
**    it
*/

      if    ( fabs( PREVIOUS_AVERAGE - AV_SCORE ) <
      STAGNATION_THRESHOLD )
            MUTATION_FACTOR += MUTATION_INCREMENT;
      else
            if    ( ( PREVIOUS_BEST - BEST_SCORE ) <
            IMPROVEMENT_REQUIREMENT)
                  MUTATION_FACTOR += MUTATION_INCREMENT;
            else
                  if    ( MUTATION_FACTOR > MUTATION_FACTOR_BASE )
                        MUTATION_FACTOR -= MUTATION_INCREMENT;

/*
**    report new mutation factor
*/

      if    ( DETAILS )
            printf( "New mutation factor = %8.6f\n", MUTATION_FACTOR );
```

```
/*
**    try a new approach - mutate between 0 and all genes depending on
**    the mutation factor
*/

      if    ( MUTATION_FACTOR > 1.0 )
            MUTATION_FACTOR = MUTN = 1.0;
      else
            MUTN = MUTATION_FACTOR;

      *MUT_FRACT = MUTN;
/*
**    for graded increase in the standard deviation of the mutation
**    operator, uncomment the next section
*/
/*

**    determine number of successive stable generations
*/
/*
**    if    ( ( PREVIOUS_BEST - BEST_SCORE ) <
      IMPROVEMENT_REQUIREMENT )
**          STABILITY_COUNT++;
**    else
**          STABILITY_COUNT = 0;
*/
/*
**    increase standard deviation linearly up to 1/3 over the number
**    of generations left in run
*/
/*
**    SD = MUTSD + (1.0/3.0 - MUTSD) * STABILITY_COUNT
**          /(MAX_GENERATION - GENERATION_NO + STABILITY_COUNT);
**
**    *MUT_SD = SD;
*/

      *MUT_SD = MUTSD;

/*
**    report mutability
*/

      PREVIOUS_AVERAGE = AV_SCORE;
      PREVIOUS_BEST = BEST_SCORE;

      if    ( DETAILS )
            printf( "Mutation potential = %8.6f\n", MUTN );

    return( MUTN );
}

/*-------------------------#########----------------------------*/
```

```
des  *Gene_To_Des(  gene_str *GENES )

/*
**   converts the gene string pointed to by GENES to a pointer to a
**   descriptor
*/

{
     int        TO, I, LAYER;
     gene_str   *TEMP;
     des        *DESCR;

/*
**   allocate memory for the new descriptor
*/

     DESCR = New( des );

/*
**   assign pointer to beginning of gene string
*/

     TEMP = GENES;

/*
**   convert each gene in the gene string into the synaptic weights
**   of inputs from one layer into each neuron
*/

     for   ( TO = 0; TO < NO_NEURONS; TO++ ) {

          for   ( LAYER = 0; LAYER < NO_LAYERS + 1; LAYER++ ) {

               for   ( I = 0; I < NO_INPUTS; I++ )
                    DESCR->WEIGHTS[TO][LAYER * NO_INPUTS + I] =
                         TEMP->GENE[I];

               TEMP = TEMP->NEXT;
          }

/*
**   assign the biasing to each neuron from the end of the gene
**   fragment for each neuron
*/

          DESCR->BIAS[TO] = TEMP->GENE[0];
          TEMP = TEMP->NEXT;
     }

/*
**   dispose of the gene string
*/

     Dispose_Gene( GENES );
```

```
/*
**    terminate the descriptor list
*/

      DESCR->NEXT = NULL;

      return( DESCR );
}
/*---------------------------##########-----------------------------*/

gene_str  *Des_To_Gene(  des  *DESCR )

/*
**    converts the descriptor pointed to by DESCR into a pointer to a
**    gene string
*/

{
      int       LAYER, TO, I;
      gene_str  *HEAD, *TEMP;

/*
**    for each neuron, create a gene containing the synaptic weights
**    of all connections with each layer
*/

      for   ( TO = 0; TO < NO_NEURONS; TO++ ) {

            for   ( LAYER = 0; LAYER < NO_LAYERS + 1; LAYER++ ) {

                  if    ( ( TO == 0 ) && ( LAYER == 0 ) )
                        TEMP = HEAD = New( gene_str );
                  else {
                        TEMP->NEXT = New( gene_str );
                        TEMP = TEMP->NEXT;
                  }

                  for   ( I = 0; I < NO_INPUTS; I++ )
                        TEMP->GENE[I] = DESCR->WEIGHTS[TO][LAYER *
                              NO_INPUTS + I];
            }

/*
**    create gene containing bias for neuron
*/

            TEMP->NEXT = New( gene_str );
            TEMP = TEMP->NEXT;

            for   ( I = 0; I < NO_INPUTS; I++ )
                  TEMP->GENE[I] = 0.0;

            TEMP->GENE[0] = DESCR->BIAS[TO];
      }
```

```
/*
**    terminate the gene string
*/

      TEMP->NEXT = NULL;

      return( HEAD );
}

/*----------------------------##########-------------------------------*/

void Swap_Gene(      gene_str   *A_GENE,
                     gene_str   *B_GENE )

/*
**    swaps genes between two gene strings
**
**    note: check you can't just reassign pointers
*/

{
      int         I;
      float       TEMP;

/*
**    swap the gene contents for each input
*/

      for   ( I = 0; I < NO_INPUTS; I++ ) {

            TEMP = A_GENE->GENE[I];
            A_GENE->GENE[I] = B_GENE->GENE[I];
            B_GENE->GENE[I] = TEMP;
      }
}

/*----------------------------##########-------------------------------*/

void Add_Details(    des   *CHILD,
                     des   *PARENT_A,
                     des   *PARENT_B,
                     int   GEN_NO,
                     int   N,
                     char BREED_CODE )

/*
**    fills in history details of newly produced child
**
**    note:       should pass in creature_no rather than descriptor
**                but i spose passing in parent allows definition of
**                identification method to remain unresolved at calling
**                level
*/
```

```
{
     CHILD->CREATURE_NO = 10000 * (long) GEN_NO + (long) N;
     CHILD->GENERATION_NO = GEN_NO;
     CHILD->PARENT_A = PARENT_A->CREATURE_NO;
     CHILD->PARENT_B = PARENT_B->CREATURE_NO;
     CHILD->BREEDING_METHOD = BREED_CODE;
     CHILD->SCORE = 0.0;
}

/*-------------------------##########-----------------------------*/

des  *Append_Des(   des *DESCR,
                    des *DESCR_LIST )

/*
**   appends the decriptor (list) pointed to by DESCR to the
**   descriptor list pointed to by DESCR_LIST i.e. adds DESCR to the
**   end of DESCR_LIST
*/

{
     des *TEMP;

     TEMP = DESCR_LIST;

/*
**   find the end of the descriptor list pointed to by TEMP and add
**   the descriptor (list) pointed to by DESCR to the end
*/

     if   ( DESCR_LIST == NULL )
          DESCR_LIST = DESCR;
     else {
          while ( TEMP->NEXT != NULL )
               TEMP = TEMP->NEXT;

          TEMP->NEXT = DESCR;
     }

     return( DESCR_LIST );
}

/*-------------------------##########-----------------------------*/

void Write_Des(     des  *DESCR,
                    FILE *OUT_FILE )

/*
**  writes a descriptor out to a file in a standard format
*/
```

```
{
     int   TO, FROM;

/*
**   identify the descriptor being written out to file
*/

     fprintf( OUT_FILE, "Creature %8ld\n", DESCR->CREATURE_NO );

/*
**   write out the synaptic weights of all connections as well as the
**   bias for each neuron
*/

     for   ( TO = 0; TO < NO_NEURONS; TO++ ) {

          for   ( FROM = 0; FROM < (NO_NEURONS + NO_INPUTS); FROM++ )
               fprintf( OUT_FILE, "%f ", DESCR->WEIGHTS[TO][FROM] );

          fprintf( OUT_FILE, "%f\n", DESCR->BIAS[TO] );
     }
}

/*---------------------------##########-----------------------------*/

int  Read_Net( weight_matrix  WEIGHTS,
               biasing        BIAS,
               FILE           *IN_FILE )

/*
**   reads in the weight and bias matrices for a network from the file
**   pointed to by IN_FILE returning a boolean indicating read success
**
**   note:     perhaps should define status type or something rather
**             than using ints for boolean flags
*/

{
     int   TO, FROM, SCAN_STAT;

     for   ( TO = 0; TO < NO_NEURONS; TO++ ) {

/*
**   read in connection weights from each neuron to each neuron,
**   exiting if error
*/

          for   ( FROM = 0; FROM < (NO_NEURONS + NO_INPUTS); FROM++ )
          {
               SCAN_STAT = fscanf( IN_FILE, "%f",
                    &WEIGHTS[TO][FROM] );
               if    ( ( SCAN_STAT == 0 ) || ( SCAN_STAT == EOF ) )
                    return( FALSE );
          }
```

```
/*
**   read in biasing for each neuron, exiting if error
*/

          SCAN_STAT = fscanf( IN_FILE, "%f", &BIAS[TO] );
          if   ( ( SCAN_STAT == 0 ) || ( SCAN_STAT == EOF ) )
               return( FALSE );
     }
     return( TRUE );
}

/*---------------------------#########-------------------------------*/

des  *Read_Networks(     int  *PTR_N )

/*
**  reads in a set of networks from files named in <run_name>.BGN
**
**  the function returns a pointer to the first network read in
**  and the number of networks read in
*/

{
     extern int     MAX_POPULATION;
     extern int     DETAILS;
     extern char    STARTIN[30];        /* contains <run_name>.BGN */

     FILE *IN_FILE, *IN_NAMES_FILE;
     char FILENAME[30];
     int  N = 0;
     int  READ_OK;
     des  *HEAD, *TEMP, *INDES;

/*
**   initialize number of networks read in
*/

     *PTR_N = 0;

/*
**  exit if <run_name>.BGN not found, or error opening
*/

     if   ( ( IN_NAMES_FILE = fopen( STARTIN, "r" ) ) == NULL )
          return( NULL );

     HEAD = NULL;

/*
**   get first input network file name
*/

     fscanf( IN_NAMES_FILE, "%s", FILENAME );
```

```
/*
**    while more input network files, and population maximum not
**    exceeded
**         open input network file
**         read first network in
**         while more networks in file
**              read network in
**         close input network file
**         get next input network file name
*/

     while ( ( feof( IN_NAMES_FILE ) == FALSE ) &&
     ( N < MAX_POPULATION ) ) {

          IN_FILE = fopen( FILENAME, "r" );
          INDES = New( des );
          READ_OK = Read_Net( INDES->WEIGHTS, INDES->BIAS, IN_FILE );

          while ( READ_OK && ( N < MAX_POPULATION ) ) {

               if    ( HEAD == NULL )
                     HEAD = TEMP = INDES;
               else {
                     TEMP->NEXT = INDES;
                      TEMP = TEMP->NEXT;
               }

               N++;

               TEMP->CREATURE_NO = (long) N;
               TEMP->GENERATION_NO = 0;
               TEMP->PARENT_A = TEMP->PARENT_B = (long) 0;
               TEMP->BREEDING_METHOD = 'I';
               TEMP->SCORE = 0.0;
               TEMP->NEXT = NULL;

               INDES = New( des );

               READ_OK = Read_Net( INDES->WEIGHTS, INDES->BIAS,
                    IN_FILE );
          }
          fclose( IN_FILE );
          fscanf( IN_NAMES_FILE, "%s", FILENAME );
     }

     fclose( IN_NAMES_FILE );
     *PTR_N = N;

     if    ( DETAILS )
          printf( "\n %d networks read in\n", N );

     return( HEAD );
}

/*--------------------------##########-----------------------------*/
```

```
void Integer_To_Bits(    unsigned  X,
                         bit_array ARRAY )

/*
**   converts the unsigned integer X into a linear array of bits as
**   wide as the network input layer
*/

{
     unsigned int   K, I;

     K = 1;

/*
**   set those locations in ARRAY which correspond to bits set in X
*/

     for  ( I = 0; I < NO_INPUTS; I++ ) {
          ARRAY[I] = (( K & X ) / K);
          K = K<<1;
     }
}

/*---------------------------##########-----------------------------*/

int  Bits_To_Integer(    bit_array ARRAY )

/*
**   converts the array of bits ARRAY into the integer returned by the
**   function
*/

{
     unsigned int   K, N, I;

     N = 0;
     K = 1;

/*
**   add the binary values represented by the ARRAY locations to N
*/

     for  ( I = 0; I < NO_INPUTS; I++ ) {

          N = N + K * ARRAY[I];
          K = K<<1;
     }

     return( N );
}

/*---------------------------##########-----------------------------*/
```

```
float     Neuron_Transf( float     IN )

/*
**   calculates a sigmoid neuron response
**
**   should be called sigmoid(in)
*/

{
     float  OUT;

     if    ( IN < 0.0 )
           OUT = 0.0;
     else {
           if    ( IN > 1.0 )
                 OUT = 1.0;
           else
                 OUT = ( ( ( AN * IN + BN ) * IN + CN ) * IN + DN) * IN
                       * IN;
     }

     return( OUT );
}

/*---------------------------##########-----------------------------*/

void Schmidt(  net_in_out     REAL,
               bit_array      BIN )

/*
**  applies a hard limiting function to the array of real numbers REAL
**   returning the array of bits BIN
*/

{
     int  I;

/*
**   set bit array location if real > 0.5 else clear bit array
**   location
*/

     for   ( I = 0; I < NO_INPUTS; I++ ) {

           if    ( REAL[I] > 0.5 )
                 BIN[I] = 1;
           else
                 BIN[I] = 0;
     }
}

/*---------------------------##########-----------------------------*/
```

```
void Bits_To_Reals( bit_array       BIN,
                    net_in_out      REAL )

/*
**  converts the bit array BIN to an array of real numbers REAL
*/

{
     int I;

/*
**  cast each bit as a real
*/

     for   ( I = 0; I < NO_INPUTS; I++ )
          REAL[I] = (float)BIN[I];
}

/*---------------------------##########------------------------------*/

int  No_Bit_Errors( bit_array NET_OUT,
                    bit_array EXPECTED_OUT )

/*
**   returns the number of bit differences between the bit array
**   NET_OUT and the bit array EXPECTED_OUT
**
**   should be called - no_bit_differ
*/

{
     int N = 0,I;

/*
**   count number of location differences
*/

     for   ( I = 0; I< NO_OUTPUTS; I++ )

          if    ( NET_OUT[I] != EXPECTED_OUT[I] )
               N++;

     return( N );
}

/*---------------------------##########------------------------------*/
```

```
float     Cumulative_Square_Error( float          ACC_SQUARE_ERROR,
                                   net_in_out     NET_OUT,
                                   net_in_out     EXPECTED_OUT )

/*
**   returns the accumulated square errors between the network output
**   NET_OUT and the expected network output EXPECTED_OUT
*/

{
     int I;

/*
**   count number of location differences
*/

     for  ( I = 0; I< NO_OUTPUTS; I++ )
          ACC_SQUARE_ERROR += (NET_OUT[I] - EXPECTED_OUT[I])
               * (NET_OUT[I] - EXPECTED_OUT[I]);

     return( ACC_SQUARE_ERROR );
}

/*-------------------------##########-------------------------------*/

des  *Generate_Rnd_Net(  int  N,
                         int  GENERATION_NO )

/*
**  generates a random network number N in generation GENERATION_NO
*/

{
     extern    float     PCON;

     des  *DESCR;
     int  TO, FROM;

/*
**   allocate space and set up descriptive information
*/

     DESCR = New( des );
     DESCR->CREATURE_NO = (long) 10000 * (long) GENERATION_NO +
          (long) N;
     DESCR->GENERATION_NO = GENERATION_NO;
     DESCR->PARENT_A = DESCR->PARENT_B = (long) 0;
     DESCR->BREEDING_METHOD = 'R';
     DESCR->NEXT = NULL;

/*
**   generate random weights for every required connection
*/
```

```
        for   ( TO = 0; TO < NO_NEURONS; TO++ ) {

              for   ( FROM = 0; FROM < (NO_NEURONS + NO_INPUTS); FROM++ )

    /*
    **    if the gaussian variable with sd=0.2, mean=0.5 < P(connection)
    **    then establish uniform random connection otherwise, don't connect
    */

                    if    ( Rand_No( 0.2, 0.5 ) < PCON )
                          DESCR->WEIGHTS[TO][FROM] = URand_No( -1, 1 );
                    else
                          DESCR->WEIGHTS[TO][FROM] = 0.0;

    /*
    **    generate a random bias
    */

              DESCR->BIAS[TO] = URand_No( -1, 1 );

        }

        return( DESCR );
    }

    /*---------------------------##########------------------------------*/

    des   *Remove_Des(    des   *ITEM,
                          des   *LIST )

    /*
    **    removes the descriptor pointed to by ITEM from the descriptor
    **    list pointed to by LIST
    */

    {
        des   *HEAD;

    /*
    **    assign ITEM->NEXT to the pointer to ITEM within LIST
    */

        HEAD = LIST;

        if    ( ITEM == LIST )
              HEAD = LIST->NEXT;
        else {

              while ( LIST->NEXT != ITEM )
                    LIST = LIST->NEXT;

              LIST->NEXT = ITEM->NEXT;
        }
```

```
/*
**    terminate the item to be returned
*/

      ITEM->NEXT = NULL;

      return( HEAD );
}

/*--------------------------#########----------------------------*/

des   *Sort(     des   *DES_LIST )

/*
**    sorts the descriptor list DES_LIST into ascending SCORE value and
**    returns a pointer to the beginning of the sorted list
*/

{
      des   *TEMP, *SORTED, *SMALLEST;

      SORTED = NULL;

/*
**    while descriptor left in DES_LIST
**          find the lowest scoring descriptor in DES_LIST
**          remove the lowest scoring descriptor from DES_LIST
**          append the descriptor removed from DES_LIST to SORTED
*/

      while ( DES_LIST != NULL ) {

            SMALLEST = TEMP = DES_LIST;

            while ( TEMP != NULL ) {

                  if    ( TEMP->SCORE < SMALLEST->SCORE )
                        SMALLEST = TEMP;

                  TEMP = TEMP->NEXT;
            }

            DES_LIST = Remove_Des( SMALLEST, DES_LIST );
            SORTED = Append_Des( SMALLEST, SORTED );
      }

      return( SORTED );
}

/*--------------------------#########----------------------------*/
```

```
void Score_BE( des   *DESCR,
               int   NUM_ERRORS )

/*
**   computes a modified score based on the number of bit errors in
**   the output, and the change in average neuron activity on the last
**   test cycle for each input presentation
**
**   note:      should use constants here
**              also, should pass in descr->delta_activation and
**              descr->activation and return a score (assuming method
**              of scoring is stable)
*/

{
     DESCR->SCORE = (float) NUM_ERRORS / (float) MAX_NO_ERRORS
          + 10.0 * DESCR->DELTA_ACTIVATION;
}

/*---------------------------##########------------------------------*/

void Score_MSE(      des         *DESCR,
                     float       MEAN_SQUARE_ERROR )

/*
**   computes a modified score based on the mean square error in the
**   output and the change in average neuron activity on the last
**   test cycle for each input presentation
**
**   note:      should use constants here
**              also, should pass in descr->delta_activation
**              and return a score (assuming method of scoring is
**              stable)
*/

{
     DESCR->SCORE = MEAN_SQUARE_ERROR / (float) MAX_NO_ERRORS
          + 10.0 * DESCR->DELTA_ACTIVATION;
}

/*---------------------------##########------------------------------*/

float     Activate_Network(   net_in_out      NET_IN,
                              net_in_out      NET_OUT,
                              weight_matrix   WEIGHTS,
                              biasing         BIAS,
                              float           *AV_ACTIVATION)

/*
**   apply the input NET_IN to the network described by WEIGHTS and
**   BIAS and return the change in neuron activation during the last
**   test cycle, the network output NET_OUT, and the average neuron
**   activation during testing in AV_ACTIVATION
*/
```

```
{
     network_state  OLD_STATE, NEW_STATE;
     int            I, J, CYCLES;
     float          TEMP, ACTIVATION;
     float          OLD_ACTIVATION = 0.0, DELTA_ACTIVATION = 1.0;

/*
**   set the initial state of the network by loading the input to the
**   network and setting all neuron responses to 0.0
*/

     for  ( I = 0; I < NO_INPUTS; I++ )
          OLD_STATE[I] = NET_IN[I];

     for  ( ; I < NO_NEURONS + NO_INPUTS; I++ )
          OLD_STATE[I] = 0.0;

/*
**   test the network until the maximum number of cycles has been
**   exceeded or the change in average activation of the network is
**   less than the minimum activation change
*/

     for  ( CYCLES = 0;
     ( ( CYCLES < CYCLE_MAX ) &&
     ( DELTA_ACTIVATION > ACTIVATION_MIN ) ); CYCLES ++ ) {

/*
**   calculate response of each neuron to excitation from every other
**   neuron in the network and the bias for the neuron
*/

          ACTIVATION = 0.0;

          for  ( I = 0; I < NO_NEURONS; I++ ) {

               TEMP = 0.0;

               for  ( J = 0; J < NO_NEURONS + NO_INPUTS; J++ )
                    TEMP += OLD_STATE[J] * WEIGHTS[I][J];

               TEMP += BIAS[I];
               NEW_STATE[I] = Neuron_Transf( TEMP );
               ACTIVATION += NEW_STATE[I];
          }

/*
**   determine average neuron activation during cycle and change in
**   activation between cycles
*/

          ACTIVATION /= ( (float) NO_NEURONS );
          DELTA_ACTIVATION = (float) fabs( (double) ( ACTIVATION -
               OLD_ACTIVATION));
          OLD_ACTIVATION = ACTIVATION;
```

```
/*
**    save the network state as initial state for next cycle
*/

          for   ( I = 0; I < NO_NEURONS; I++ )
                OLD_STATE[I + NO_INPUTS] = NEW_STATE[I];
     }

/*
**    set the network response to be returned
*/

     for   ( I = 0; I < NO_INPUTS; I++ )
          NET_OUT[I] = OLD_STATE[I + NO_NEURONS];

/*
**    set the network activation to be returned as the average
**    activation during the last cycle
*/

     *AV_ACTIVATION = ACTIVATION;

/*
**    return the average change in activation during the last cycle
*/

     return( DELTA_ACTIVATION );
}

/*----------------------------##########-----------------------------*/

void Test(      des   *DESCR )

/*
**  test the network pointed to by DESCR
*/

{
     float      ACT_SUM = 0.0, DELTA_SUM = 0.0, ACT;
     int        N = 0, IN;
     net_in_out NET_IN, NET_OUT;
     bit_array BINARY_IN, EXPECTED_BIN_OUT;

/*
**    if scoring bit errors, uncomment following lines
**         and comment lines relating to mean square errors
*/
/*
**    bit_array BINARY_OUT;
**    int       NUM_ERRORS = 0;
*/
/*
**    end bit errors
*/
```

```
/*
**   if scoring mean square errors, uncomment following lines
**        and comment lines relating to bit errors
*/
     float          ACCUM_ERROR = 0.0;
     net_in_out     EXPECTED_REAL_OUT;
/*
**   end mean square errors
*/
/*
**   for each input to be presented
**        convert it to an input array
**        present the input to the network
**        sum the last change in activation and average activation
**        sum the number of bit errors in the network output
*/

     for  ( IN = TEST_MIN; IN < TEST_MAX; IN += TEST_STEP ) {

          N++;
          Integer_To_Bits( training_set[IN][TRAIN_IN], BINARY_IN );
          Bits_To_Reals( BINARY_IN, NET_IN );
          DELTA_SUM += Activate_Network( NET_IN, NET_OUT,
               DESCR->WEIGHTS, DESCR->BIAS, &ACT );
          ACT_SUM += ACT;
          Integer_To_Bits( training_set[IN][TRAIN_OUT],
               EXPECTED_BIN_OUT );

/*
**   if scoring bit errors, uncomment the following lines
**        and comment lines relating to mean square errors
*/
/*
**        Schmidt(NET_OUT,BINARY_OUT);
**        NUM_ERRORS += No_Bit_Errors(BINARY_OUT, EXPECTED_BIN_OUT);
*/
/*
**   end bit errors
*/
/*
**   if scoring mean square errors, uncomment the following lines
**        and comment lines relating to bit errors
*/

          Bits_To_Reals( EXPECTED_BIN_OUT, EXPECTED_REAL_OUT );
          ACCUM_ERROR = Cumulative_Square_Error( ACCUM_ERROR, NET_OUT,
               EXPECTED_REAL_OUT );
/*
**   end mean square errors
*/
     }

/*
**   record the network activation performance
*/
```

```
        DESCR->ACTIVATION = ACT_SUM / ( (float) N );
        DESCR->DELTA_ACTIVATION = DELTA_SUM / ( (float) N );

    /*
    **    score the network performance
    */
    /*
    **    if scoring bit errors, uncomment following lines
    **         and comment lines relating to mean square errors
    */
    /*
    **    Score_BE( DESCR, NUM_ERRORS );
    */
    /*
    **    end bit errors
    */
    /*
    **    if scoring mean square errors, uncomment the following lines
    **         and comment lines relating to bit errors
    */
        Score_MSE( DESCR, (float) sqrt( (double) ACCUM_ERROR ) );
    /*
    **    end mean square errors
    */
    }

    /*----------------------------#########-------------------------------*/

    void Rate(      des         *DES_LIST,
                    float       *BEST_SCORE,
                    float       *AVERAGE_SCORE )

    /*
    **    function to test the list of descriptors pointed to by DES_LIST
    **    returning the best and average score of all descriptors
    **
    **    note:     BEST should be a function of constants used in Score()
    */

    {
        extern    int  DETAILS;

        int         N = 0;
        float       BEST = 4.0, SUM = 0.0;

    /*
    **    while more descriptors
    **         display details of network being tested if required
    **         test the network
    **         save the score if best
    **         accumulate the score to determine average
    **         reset pointer to the next descriptor in the list
    */
```

```
        while ( DES_LIST  != NULL ) {

                if    ( DETAILS )
                      printf( "Testing creature %4d of generation %4d\n",
                            N, DES_LIST->GENERATION_NO );

                Test( DES_LIST );

                N++;

                if    ( DES_LIST->SCORE < BEST )
                      BEST = DES_LIST->SCORE;

                SUM += DES_LIST->SCORE;
                DES_LIST = DES_LIST->NEXT;
        }

    /*
    **    return the average and best scores for the list
    */

        *AVERAGE_SCORE = SUM / ( (float) N );
        *BEST_SCORE = BEST;
    }

    /*----------------------------##########-------------------------------*/

    des   *Breed(    des   *PARENT_A,
                     des   *PARENT_B,
                     int   GENERATION_NO,
                     int   N )

    /*
    **    function to return a pointer to a list of child descriptors
    **    numbered from N produced at generation GENERATION_NO by
    **    'breeding' the descriptors pointed to by PARENT_A and PARENT_B
    **
    **    note:       N should be returned as a parameter because we don't
    **                know at the calling level how many children will be
    **                produced
    */

    {
        des         *CHILD1, *CHILD2;
        gene_str    *A_GENE, *B_GENE, *TEMP_A, *TEMP_B;

    /*
    **    convert the two parent descriptors to gene strings
    */

        TEMP_A = A_GENE = Des_To_Gene( PARENT_A );
        TEMP_B = B_GENE = Des_To_Gene( PARENT_B );
```

```
/*
**    swap genes between the two strings at random with P(swap) = 0.5
*/

      while ( TEMP_A != NULL ) {

            if    ( Rand_No(0.2,0.5) < 0.5 )
                  Swap_Gene( TEMP_A, TEMP_B );

            TEMP_A = TEMP_A->NEXT;
            TEMP_B = TEMP_B->NEXT;
      }

/*
**    convert the two gene strings back into descriptors representing
**    the children and add the details of their ancestry
*/

      CHILD1 = Gene_To_Des( A_GENE );
      Add_Details( CHILD1, PARENT_A, PARENT_B, GENERATION_NO, N,
            'N' );
      CHILD2 = Gene_To_Des( B_GENE );
      Add_Details( CHILD2, PARENT_A, PARENT_B, GENERATION_NO, N+1,
            'N');

/*
**    convert the two child descriptors into a terminated list of
**    descritors and return the pointer to the beginning of the list
*/

      CHILD1 -> NEXT = CHILD2;
      CHILD2 -> NEXT = NULL;

      return( CHILD1 );
}

/*---------------------------#########-------------------------------*/

des   *Mutate(  des        *PARENT,
                float      MUT_FRACT,
                float      MUT_SD,
                int        GENERATION_NO,
                int        N )

/*
**    returns a pointer to a mutation having number N at generation
**    GENERATION_NO of the descriptor pointed to by PARENT. The
**    "severity of mutation" or "propensity for mutation" or "mutation
**    potential" or "mutability" is represented by MUTATN
**
**    IMPLEMENTS MUTATION STRATEGY
*/
```

```
{
     des       *MUTANT;
     gene_str  *M_GENE, *TEMP_G_PTR;
     int       I;
     short     mutation_indicated[NO_GENES], initial, final;
     int       j, num_mutations;

/*
**   convert the parent description into a gene string
*/

     TEMP_G_PTR = M_GENE = Des_To_Gene( PARENT );

/*
**   for each gene in the gene string
**        if it should be mutated
**             modify each code in the gene in some random way
*/

/*
**   determine number of genes to be mutated
*/

     if   ( MUT_FRACT > 0.5 ) {
          num_mutations = NO_GENES * ( 1.0 - MUT_FRACT );
          initial = TRUE;
          final = FALSE;
     }
     else {
          num_mutations = NO_GENES * MUT_FRACT;
          initial = FALSE;
          final = TRUE;
     }

     for  ( j=0; j<NO_GENES; j++ )
          mutation_indicated[j] = initial;

/*
**   determine which genes are to be mutated
*/

     for  ( j=0; j<num_mutations; j++ )
          mutation_indicated[Urand( NO_GENES )] = final;

     j = 0;

/*
**   for each gene to be mutated
**        add a gaussian variable to the connection weights
**             ensuring weight stays in range [-1, 1]
*/
```

```
        while ( TEMP_G_PTR != NULL ) {

                if    ( mutation_indicated[j] )
                      for  ( I = 0; I < NO_INPUTS; I++ ) {
                           TEMP_G_PTR->GENE[I] =
                                fabs( (double) Rand_No( MUT_SD, 0.0 ) )
                                + TEMP_G_PTR->GENE[I] + 1.0;
                           if   ( TEMP_G_PTR->GENE[I] < 2.0 )
                                TEMP_G_PTR->GENE[I] -= 1.0;
                           else
                                TEMP_G_PTR->GENE[I] = -1.0 *
                                     TEMP_G_PTR->GENE[I] + 3.0;
                      }

                TEMP_G_PTR = TEMP_G_PTR->NEXT;

                ++j;

        }

    /*
    **   convert the mutant gene string back into a descriptor,
    **   add the ancestry details to the descriptor,
    **   terminate the descriptor list at the mutant,
    **   and return the pointer to the mutant
    */

        MUTANT = Gene_To_Des( M_GENE );
        Add_Details( MUTANT, PARENT, PARENT, GENERATION_NO, N, 'm' );
        MUTANT->NEXT = NULL;

        return( MUTANT );
    }

    /*---------------------------#########------------------------------*/

    void Report(   des       *PARENTS,
                   float     BEST_SCORE,
                   float     AV_SCORE,
                   int       NO_SURVIVORS,
                   int       NO_GENERATED,
                   int       GENERATION_NO,
                   float     MUT_SD,
                   float     MUT_FRACT )

    /*
    **   reports on the performance of the descriptors pointed to by
    **   PARENTS
    */
```

```
{
     extern int     DETAILS;
     extern char    DESCOUT[30];
     extern char    GENOUT[30];
     extern char    SUMOUT[30];

     des       *TEMP;
     FILE      *OUT_FILE;
     time_t    CURRENT_TIME;
     struct    tm   *timeptr;

     time( &CURRENT_TIME );
     timeptr = localtime( &CURRENT_TIME );

/*
**   display which generation is being reported on if required
*/

     if   ( DETAILS )
          printf( "Reporting on generation %4d\n", GENERATION_NO );

/*
**   create the report files if reporting on the initial generation
*/

     if   ( GENERATION_NO == 0 ) {
          OUT_FILE = fopen( DESCOUT, "w");
          fclose( OUT_FILE );
          OUT_FILE = fopen( SUMOUT, "w" );
          fprintf( OUT_FILE, " Gen  Best      Average   Num    Num"
               "    DateTime  Mutation  Mutation\n" );
          fprintf( OUT_FILE, " Num  Score     Score     Gend   Surv"
               "   %02d-%02d-%2d  Fraction  Std Dev.\n",
               timeptr->tm_mday, (timeptr->tm_mon)+1,
               timeptr->tm_year);
     }

/*
**   to append the descriptions of any networks not already reported
**   on to the network description file <run_name>.DSC, uncomment
**   the following lines
*/
/*
**   TEMP = PARENTS;
**
**   OUT_FILE = fopen( DESCOUT, "a" );
**
**   while ( TEMP != NULL ) {
**
**        if   ( ( TEMP->GENERATION_NO == GENERATION_NO )
**        &&   ( TEMP->BREEDING_METHOD != 'I' ) )
**             Write_Des( TEMP, OUT_FILE );
**
**        TEMP = TEMP->NEXT;
**    }
```

```
**
**    fclose( OUT_FILE );
*/
/*
**    to append the results of this generation's best performers to the
**    generation results file <run_name>.GEN, uncomment the following
**    lines
*/
/*
**    OUT_FILE = fopen( GENOUT, "a" );
**
**    fprintf( OUT_FILE, "Generation No : %d \n No. Generated : %d\n""
**         " No. SURVIVORS : %d\n", GENERATION_NO, NO_GENERATED,
**         NO_SURVIVORS );
**     fprintf( OUT_FILE, "Creature No.  Parent 1  Parent 2  Breeding"
**         " Score" );
**    fprintf( OUT_FILE, "   Activation   D_act    #Err\n" );
**
**    TEMP = PARENTS;
**
**    while ( TEMP != NULL ) {
**
**         if    ( TEMP->GENERATION_NO == GENERATION_NO )
**              fprintf( OUT_FILE, " %8ld  %8ld  %8ld            %c"
**                   "  %8.6f  %8.6f  %8.6f  %2d\n", TEMP->CREATURE_NO,
**                   TEMP->PARENT_A, TEMP->PARENT_B,
**                   TEMP->BREEDING_METHOD, TEMP->SCORE,
**                   TEMP->ACTIVATION, TEMP->DELTA_ACTIVATION,
**                   DisplayTest( TEMP ) );
**
**         TEMP = TEMP->NEXT;
**    }
**
**    fclose( OUT_FILE );
*/
/*
**    append a one line summary for this generation to the summary file
**    <run_name>.SUM
*/


     OUT_FILE = fopen( SUMOUT, "a" );


     fprintf( OUT_FILE, "%4d  %8.6f  %8.6f  %4d   %4d  "
          " %02d:%02d:%02d  %8.6f  %8.6f\n",
          GENERATION_NO, BEST_SCORE, AV_SCORE, NO_GENERATED,
          NO_SURVIVORS, timeptr->tm_hour, timeptr->tm_min,
          timeptr->tm_sec, MUT_FRACT, MUT_SD );

     fclose( OUT_FILE );
}

/*-------------------------##########-----------------------------*/
```

```
des  *Create_Generation( des        *PARENTS,
                         int        *NO_GENERATED,
                         int        GENERATION_NO,
                         float      MUT_FRACT,
                         float      MUT_SD )

/*
**   creates the new generation GENERATION_NO by generating
**   NO_GENERATED mutant and normal children from the descriptors
**   pointed to by PARENTS
*/

{
     extern    int  DETAILS;

     des  *PARENT_A, *PARENT_B, *CHILDREN, *CHILD;
     int  N = 0;

/*
**   create pointers to the PARENTS and CHILDREN
*/

     PARENT_A = PARENT_B = PARENTS;
     CHILDREN = NULL;

/*
**   for each parent
**        pair it with every subsequent parent in the list including
**             itself
**        if paired with itself
**             create a mutant child
**        else
**             breed normal children
*/

     while ( PARENT_A != NULL ) {

          while ( PARENT_B != NULL ) {

               N++;

               if   ( DETAILS )
                    printf( "Creating creature %4d of generation"
                         " %4d\n", N, GENERATION_NO );

               if   ( PARENT_A == PARENT_B )
                    CHILD = Mutate( PARENT_A, MUT_FRACT,
                         MUT_SD, GENERATION_NO, N );
               else {
                    CHILD = Breed( PARENT_A, PARENT_B, GENERATION_NO,
                         N );
                    N++;
               }
```

```
                CHILDREN = Append_Des( CHILDREN, CHILD );
                PARENT_B = PARENT_B -> NEXT;
            }

            PARENT_A = PARENT_A -> NEXT;
            PARENT_B = PARENT_A;
        }

    /*
    **  create a composite list of CHILDREN and PARENTS
    */

        CHILDREN = Append_Des( CHILDREN, PARENTS );

    /*
    **  return the number generated
    */

        *NO_GENERATED = N;

    /*
    **  return the pointer to the CHILDREN
    */

        return( CHILDREN );
    }

    /*---------------------------##########-----------------------------*/

    des  *Select(  des      *CHILDREN,
                   int      *NO_SURVIVORS,
                   float    BEST_SCORE,
                   float    AV_SCORE,
                   int      GENERATION_NO )

    /*
    **  returns a pointer to the surviving CHILDREN as well as their
    **  count in NO_SURVIVORS based on the BEST_SCORE, AV_SCORE,
    **  MIN_POPULATION, and MAX_POPULATION
    */

    {
        extern int      DETAILS;
        extern int      MAX_POPULATION;
        extern int      MIN_POPULATION;

        des     *PTR;
        int     N = 0;
        float   MAX_DEV;

    /*
    **  display which generation is being sorted if required and
    **  sort the CHILDREN into increasing scores
    */
```

```
    if    ( DETAILS )
          printf( "Sorting creatures for generation %4d\n",
               GENERATION_NO );

    PTR = CHILDREN = Sort( CHILDREN );

/*
**  determine the maximum deviation from best score to select as a
**  surviving score
*/

    MAX_DEV = ( AV_SCORE - BEST_SCORE ) * 0.2 + BEST_SCORE;

/*
**  report commencement of selection if required
*/

    if    ( DETAILS )
          printf( "Selecting survivors for generation %4d\n",
               GENERATION_NO);

/*
**  keep selecting survivors from amongst the children as long as at
**  least MIN_POPULATION are selected, and no more than
**  MAX_POPULATION are selected, and their score deviates less than
**  MAX_DEV from the best score
*/

    while ( ( N < MIN_POPULATION )
    || ( ( N < MAX_POPULATION )
    && ( CHILDREN->SCORE < MAX_DEV )
    && ( CHILDREN->NEXT != NULL ) ) ) {
          CHILDREN = CHILDREN->NEXT;
          N++;
    }

/*
**  dispose of the unselected children, freeing up spces occupied by
**  them, terminate the list of survivors, and return the number of
**  survivors as well as a pointer to the survivors
*/


    Dispose_Des( CHILDREN->NEXT );
    CHILDREN->NEXT = NULL;
    *NO_SURVIVORS = N;

    return( PTR );
}

/*--------------------------#########-------------------------------*/
```

```
void DisplayList(   des   *DESCRIPTOR_LIST )

/*
**   displays the descriptors pointed to by DESCRIPTOR_LIST on the
**   screen
*/

{
     des   *TEMP;
     int   N = 0;

     TEMP = DESCRIPTOR_LIST;

/*
**   while more descriptors
**        display the descriptor on the screen
*/

     while ( TEMP != NULL ) {
          N++;
          printf( "\nDisplay of creature %4d in generation %4d\n",
               N, TEMP->GENERATION_NO );
          printf( "-----------------------------------------\n\n");
          DisplayDescriptor( TEMP );
          TEMP = TEMP->NEXT;
    }
}

/*-------------------------##########-----------------------------*/

void DisplayDescriptor(  des *DESCRIPTOR )

/*
**  displays the descriptor pointed to by DESCRIPTOR on the screen
*/

{
     int   TO, FROM;

/*
**   display the descriptor details
*/

     printf( "Creature number = %ld\n", DESCRIPTOR->CREATURE_NO );
     printf( "Generation number = %4d\n", DESCRIPTOR->GENERATION_NO);
     printf( "Parent A = %ld\n", DESCRIPTOR->PARENT_A );
     printf( "Parent B = %ld\n", DESCRIPTOR->PARENT_B );
     printf( "Breeding method = %c\n", DESCRIPTOR->BREEDING_METHOD );

/*
**   for each neuron in the network
**        display the connection weights from every other neuron
**        display the bias of the neuron
*/
```

```
        for   ( TO = 0; TO < NO_NEURONS; TO++ ) {

              for   ( FROM = 0; FROM < (NO_NEURONS + NO_INPUTS); FROM++ )
                   printf( "Weight [%4d][%4d] = %8.6f\n",
                        TO, FROM, DESCRIPTOR->WEIGHTS[TO][FROM] );

              printf("Bias [%4d] = %8.6f\n\n",TO,DESCRIPTOR->BIAS[TO]);
        }

    /*
    **   display the performance of the network
    */

        printf( "Score = %9.6f\n", DESCRIPTOR->SCORE );
        printf( "Activation = %9.6f\n", DESCRIPTOR->ACTIVATION );
        printf( "Delta activation = %9.6f\n",
             DESCRIPTOR->DELTA_ACTIVATION );

    /*
    **   display the test results for the network
    */

        DisplayTest( DESCRIPTOR );
    }

    /*--------------------------#########----------------------------*/

    int  DisplayTest(   des  *DESCRIPTOR )

    /*
    **   display the test results for the network pointed to by DESCRIPTOR
    **   and return the number of errors made by the network in all tests
    */

    {
        float          ACTIV;
        int            IN, NUM_ERRORS = 0;
        net_in_out     NET_IN, NET_OUT;
        bit_array      BINARY_IN, BINARY_OUT, EXPECTED_OUT;

    /*
    **   printf( "Input Expect    Output\n" );
    **   printf( "----- ------    ------\n" );
    */

    /*
    **   for each test
    **        derive the input to be presented to the network
    **        test the network
    **        sum the number of errors made in the output
    */
```

```
        for   ( IN = TEST_MIN; IN < TEST_MAX; IN += TEST_STEP ) {
              Integer_To_Bits( training_set[IN][TRAIN_IN], BINARY_IN );
              Bits_To_Reals( BINARY_IN, NET_IN );
              Activate_Network( NET_IN, NET_OUT, DESCRIPTOR->WEIGHTS,
                   DESCRIPTOR->BIAS, &ACTIV );
              Schmidt( NET_OUT, BINARY_OUT );
              Integer_To_Bits( training_set[IN][TRAIN_OUT], EXPECTED_OUT);
              NUM_ERRORS += No_Bit_Errors( BINARY_OUT, EXPECTED_OUT );
    /*
    **        printf("%5d    %6d  %6d\n", IN, F(IN),
    **             Bits_To_Integer( BINARY_OUT ) );
    */
        }

    /*
    **  return the number of errors made
    */

        return( NUM_ERRORS );
    }

    /*---------------------------#########-----------------------------*/

    des  *Generate(     des       *PARENTS,
                        int       *NO_GENERATED,
                        int       GENERATION_NO,
                        float     MUT_FRACT,
                        float     MUT_SD,
                        float     *BEST_SCORE,
                        float     *AV_SCORE,
                        float     *WORST_SCORE,
                        int       *NO_SURVIVORS )

    /*
    **  creates the new generation GENERATION_NO by generating
    **  NO_GENERATED mutant and normal children from the descriptors
    **  pointed to by PARENTS
    */

    {
        extern int     DETAILS;
        extern int     MAX_POPULATION;
        extern int     MIN_POPULATION;

        des       *PARENT_A, *PARENT_B, *CHILDREN, *CHILD, *TEMP, *CP,
                  *WORST_CHILD;
        float     SCORE_SUM = 0.0, CHILD_SCORE, MAX_DEV,
                  WORST_CHILD_SCORE;
        int       NG = 0, NS = 0, NC = 0;

    /*
    **  create pointers to the PARENTS and CHILDREN
    */
```

```
        PARENT_A = PARENT_B = PARENTS;
        CHILDREN = NULL;

    /*
    **  for each parent
    **      pair it with every subsequent parent in the list including
    **          itself
    **      if paired with itself
    **          create a mutant child
    **      else
    **          breed normal children
    */

        while ( PARENT_A != NULL ) {

            while ( PARENT_B != NULL ) {

                NG++;

                if   ( DETAILS )
                    printf( "Creating creature %4d of generation"
                        " %4d\n", NG, GENERATION_NO );

                if   ( PARENT_A == PARENT_B ) {
                    CHILD = Mutate( PARENT_A, MUT_FRACT, MUT_SD,
                        GENERATION_NO, NG );
                    SCORE_SUM += PARENT_A->SCORE;
                }
                else {
                    CHILD = Breed( PARENT_A, PARENT_B, GENERATION_NO,
                        NG );
                    NG++;
                }

                TEMP = CHILD;

                while ( TEMP != NULL ) {
                    CHILD = CHILD->NEXT;
                    TEMP->NEXT = NULL;
                    Test( TEMP );
                    CHILD_SCORE = TEMP->SCORE;
                    SCORE_SUM += CHILD_SCORE;
                    if   ( CHILD_SCORE < *BEST_SCORE )
                        *BEST_SCORE = CHILD_SCORE;
                    if   ( CHILD_SCORE < *WORST_SCORE )
                        if   ( NC < MAX_POPULATION )
                            if   ( CHILDREN == NULL ) {
                                CHILDREN = TEMP;
                                WORST_CHILD_SCORE = CHILD_SCORE;
                                WORST_CHILD = TEMP;
                                NC++;
                            }
                            else
```

```
                                    if    ( CHILD_SCORE >=
                                    WORST_CHILD_SCORE ) {
                                          WORST_CHILD->NEXT = TEMP;
                                          WORST_CHILD = TEMP;
                                          WORST_CHILD_SCORE    =
                                               CHILD_SCORE;
                                          NC++;
                                    }
                                    else
                                          if    ( CHILDREN->SCORE >=
                                          CHILD_SCORE ) {
                                                TEMP->NEXT = CHILDREN;
                                                CHILDREN = TEMP;
                                                NC++;
                                          }
                                          else {
                                                CP = CHILDREN;
                                                while ( CP->NEXT->SCORE
                                                < CHILD_SCORE )
                                                     CP = CP->NEXT;
                                                TEMP->NEXT = CP->NEXT;
                                                CP->NEXT = TEMP;
                                                NC++;
                                          }
                              else
                                    if    ( CHILD_SCORE < WORST_CHILD_SCORE)
                                    {
                                          CP = CHILDREN;
                                          if    ( CHILDREN->SCORE >=
                                          CHILD_SCORE ) {
                                                TEMP->NEXT = CHILDREN;
                                                CHILDREN = TEMP;
                                          }
                                          else {
                                                while ( CP->NEXT->SCORE
                                                < CHILD_SCORE )
                                                     CP = CP->NEXT;
                                                TEMP->NEXT = CP->NEXT;
                                                CP->NEXT = TEMP;
                                          }
                                          while ( CP->NEXT->NEXT != NULL )
                                                CP = CP->NEXT;
                                          WORST_CHILD_SCORE = CP->SCORE;
                                          WORST_CHILD = CP;
                                          Free( CP->NEXT );
                                          CP->NEXT = NULL;
                                    }
                                    else
                                          Free( TEMP );
                        else
                              Free( TEMP );
                        TEMP = CHILD;
                  }
```

```
            PARENT_B = PARENT_B -> NEXT;
            }

            PARENT_A = PARENT_A -> NEXT;
            PARENT_B = PARENT_A;
      }

/*
**    create a composite list of CHILDREN and PARENTS
*/

      CHILDREN = Append_Des( PARENTS, CHILDREN);

      *AV_SCORE = SCORE_SUM / (*NO_SURVIVORS + NG);

      TEMP = CHILDREN = Sort( CHILDREN );

/*
**    determine the maximum deviation from best score to select as a
**    surviving score
*/

      MAX_DEV = ( *AV_SCORE - *BEST_SCORE ) * 0.2 + *BEST_SCORE;

/*
**    report commencement of selection if required
*/

      if    ( DETAILS )
            printf( "Selecting survivors for generation %4d\n",
                  GENERATION_NO);

/*
**    keep selecting survivors from amongst the children as long as at
**    least MIN_POPULATION are selected, and no more than
**    MAX_POPULATION are selected, and their score deviates less than
**    MAX_DEV from the best score
*/

      while ( ( NS < MIN_POPULATION )
      || ( ( NS < MAX_POPULATION )
      && ( CHILDREN->SCORE < MAX_DEV )
      && ( CHILDREN->NEXT != NULL ) ) ) {
            WORST_CHILD = CHILDREN;
            CHILDREN = CHILDREN->NEXT;
            NS++;
      }

/*
**    dispose of the unselected children, freeing up space occupied by
**    them, terminate the list of survivors, and return the number of
**    survivors as well as a pointer to the survivors
*/
```

```
        Dispose_Des( CHILDREN );
        WORST_CHILD->NEXT = NULL;
        *NO_SURVIVORS = NS;
        *WORST_SCORE = WORST_CHILD->SCORE;

        CHILDREN = TEMP;

    /*
    **  return the number generated
    */

        *NO_GENERATED = NG;

    /*
    **  return the pointer to the CHILDREN
    */

        return( CHILDREN );
    }

    /*---------------------------##########-------------------------------*/
```

```
/*****************************************************************
**                                                              **
**    filename:       random.h                                  **
**    programmer:     o.diessel                                 **
**    modified:       901021ofd                                 **
**    description:    Contains prototypes for random functions  **
**                                                              **
*****************************************************************/

float     Rand_No(        float     stddev,
                          float     mean );
/*
**    returns a random number with expected value, mean, and standard
**    deviation, stddev
*/

/*---------------------------#########-------------------------------*/

void      Randomize(      unsigned  seed );

/*
**    seeds a new random number
*/

/*---------------------------#########-------------------------------*/

int       Urand(          int       max );

/*
**    returns a uniformly distributed random number between 0 and max-1
*/

/*---------------------------#########-------------------------------*/

float     URand_No(       float     min,
                          float     max );

/*
**    returns uniformly distributed random variable between min and max
*/

/*---------------------------#########-------------------------------*/
```

```
/*****************************************************************
**                                                              **
**    filename:       random.c                                  **
**    programmer:     m.bentink                                 **
**    modified:       901019ofd                                 **
**    description:    contains random variable functions        **
**                                                              **
**                                                              **
*****************************************************************/

#include  <stdio.h>

/*
**    TurboC specifics - comment out for run on Apollo
*/

#include  <stdlib.h>

#define   RAND_MAX    0x7FFF

/*
**    Apollo specifics - comment out for run on PC
*/
/*    #define   RAND_MAX  2147483647
*/

/*----------------------------#########-------------------------------*/

float     Rand_No(  float     SD,
                    float     MU )

/*
**    function to generate a normally distributed random number with
**    mean: MU and standard deviation: SD
**
**    ref: Muller, Marvin E. "A Comparison of Methods for Generating
**    Normal Deviates on Digital Computers", Journal of the A.C.M., VI
**    (1959), 376-383.
*/

{
     float     SUM = 0.0;
     int       I;

     for   ( I = 0; I < 12; I++ )
          SUM += (float)rand()/RAND_MAX;

     return( SD * ( SUM - 6.0 ) + MU );
}

/*----------------------------#########-------------------------------*/
```

```
void Randomize(     unsigned  SEED )

/*
**    seed new random number
*/

{
     srand( SEED );
}

/*----------------------------##########-----------------------------*/

int  Urand(    int  max )

/*
**    returns a uniformly distributed random number between 0 and max-1
*/

{
     return( (int) ( (float) rand() /
          (float) RAND_MAX * (float) max) );
}

/*----------------------------##########-----------------------------*/

float     URand_No( float     min,
                    float     max )

/*
**    function to generate a uniformly distributed random number
**    ranging between min and max
*/

{
     return( min + (float) rand() /
          (float) RAND_MAX * ( max - min ) );
}

/*----------------------------##########-----------------------------*/
```

```
/*****************************************************************
**                                                              **
**   filename:       memory.h                                   **
**   programmer:     o.diessel                                  **
**   modified:       901016ofd                                  **
**   description:    contains prototypes and macro definitions of **
**                   memory functions                           **
**                                                              **
*****************************************************************/

/*
**   Memory prototypes
*/

void       Memory_Error(  void );

char       *New_Memory(   unsigned  size );

void       Free_Memory(   char      *memptr,
                          int       size );

/*
**   Memeory macro definitions
*/

#define    New(ptr_type)   (ptr_type *)New_Memory(sizeof(ptr_type))

#define    Free(ptr)       Free_Memory((char *)ptr,sizeof(*ptr))
```

```
/*****************************************************************
**                                                              **
**    filename:       memory.c                                  **
**    programmer:     m.bentink                                 **
**    modified:       901016ofd                                 **
**    description:    contains functions to allocate and deallocate**
**                    memory and keep track of memory usage     **
**                                                              **
*****************************************************************/

#include  <stdio.h>

/*
**    TurboC specifics - comment out for run on Apollo
*/
#include  <stdlib.h>

/*
**    Definitions
*/

#define    MEM_INCREMENT  131072L    /* 100k memory increment */

/*
**    Global variables
*/

long MEMORY_USAGE = 0;
long PEAK = MEM_INCREMENT;

/*----------------------------##########-----------------------------*/

void Memory_Error(  void )

/*
**    report fatal memory error and stop
*/

{
     printf( "OUT OF MEMORY ERROR\n" );
     exit( 1 );
}

/*----------------------------##########-----------------------------*/

char *New_Memory(   unsigned  SIZE )

/*
**    allocate SIZE bytes of memory
*/

{
     char *PTR;
```

```
/*
**    update record of memory usage
*/

      MEMORY_USAGE += (long) SIZE;

/*
**    display memory usage whenever an increment in memory usage
**    is made
*/

      if    ( MEMORY_USAGE > PEAK ) {
            printf( "New peak working set : %ld Kbytes\n",
                  MEMORY_USAGE / (long) 1024 );
            PEAK += MEM_INCREMENT;
      }

/*
**    allocate memory and report error if not done
*/

      PTR = (char *) malloc( SIZE );

      if    ( PTR == NULL )
            Memory_Error();

/*
**    return pointer to memory
*/

    return( PTR );
}

/*---------------------------#########-------------------------------*/

void Free_Memory(    char       *PTR,
                     int        SIZE )

/*
**    frees the block of memory pointed to by PTR and updates
**    MEMORY_USAGE by SIZE
*/

{
      free( PTR );
      MEMORY_USAGE -= (long) SIZE;
}

/*---------------------------#########-------------------------------*/
```

```
/*****************************************************************
**                                                              **
**   filename:      scrnplot.h                                  **
**   programmer:    o.diessel                                   **
**   modified:      901021ofd                                   **
**   description:   Contains prototypes for screen plotting     **
**                  functions                                   **
**                                                              **
*****************************************************************/

void InitializeScores(   void );

void AddToScoreList(     int       generation_number,
                         float     best_score,
                         float     average_score );

void PlotScores(         void );

void DisposeOfScores(    void );
```

```
/*****************************************************************
**                                                              **
**   filename:      scrnplot.c                                  **
**   programmer:    o.diessel                                   **
**   modified:      901021ofd                                   **
**   description:   Contains functions to plot scores on screen **
**                                                              **
*****************************************************************/
#include  <stdio.h>
#include  <string.h>
#include  <math.h>

/*
**   TurboC specifics - comment out to run on Apollo
*/

#include  <time.h>

/*
**   Apollo specifics - comment out to run on PC
*/
/*   #include  <sys/types.h>
*/

#include  "memory.h"
#include  "scrnplot.h"

#define   MAX_SCORE 100.0
#define   MIN_SCORE -100.0

typedef   struct    sl   {    int       generation_number;
                              float     best_score;
                              float     average_score;
                              time_t    completion_time;
                              struct sl *gen_next;  /* ascending */
                              struct sl *best_next; /* descending*/
                              struct sl *avg_next; ' /* descending*/
                         } score_list;


typedef   struct    rl   {    int       gen_number;
                              char      marker[20];
                              struct rl *next;
                         } range_list;

struct    score_head     {    int       score_list_length;
                              struct    bs   {    float     max;
                                                 float     min;
                                             } best_score;
                              struct    as   {    float     max;
                                                 float     min;
                                             } average_score;
                              time_t    last_insert;
                              score_list *gen_first; /* ascending*/
                              score_list *gen_last;
```

```
                                score_list *best_first; /* descending*
                                score_list *avg_first; /* descending*/
                            };

struct    score_head    scores;

/*
**   Local prototypes
*/

range_list     *AddToRangeList(    range_list     *rl,
                                   int            gen_num,
                                   char           *marker );

void           PlotRangeList(      range_list     *rl,
                                   float          dpoint );

void           DisposeOfRangeList( range_list     *rl );

/*----------------------------#########-----------------------------*/

range_list     *AddToRangeList(    range_list     *rl,
                                   int            gen_num,
                                   char           *marker )

/*
**   adds range_list element to range list rl in generation number
**   order returning pointer to new range_list
*/

{
     range_list     *temp, *crp, *prp;

/*
**   create new element
*/

     temp = New( range_list );

/*
**   set details
*/

     temp->gen_number = gen_num;
     strcpy( temp->marker, marker );
     temp->next = NULL;
```

```
/*
**    if list empty
**         add to front of list
**    else
**         if generation lowest
**              add to front
**         else
**              search for generation in list which is greater, and
**              insert in front
*/

     if    ( rl == NULL )
           rl = temp;
     else {
           if    ( rl -> gen_number > gen_num ) {
                 temp->next = rl;
                 rl = temp;
           }
           else {
                 crp = prp = rl;

                 while ( crp != NULL && crp->gen_number < gen_num ) {
                       prp = crp;
                       crp = crp->next;
                 }

                 if    ( crp == NULL || crp->gen_number > gen_num ) {
                       temp->next = crp;
                       prp->next = temp;
                 }
           }
     }
     return( rl );
}

/*---------------------------##########-------------------------------*/

void PlotRangeList( range_list      *rl,
                    float           dpoint )

/*
**    plots range_list rl with domain spacing dpoint
*/

{
     range_list      *temp;
     float           f=dpoint;

     temp = rl;
```

```
/*
**    while pointing to an element in the list
**         if domain value > generation number for the current element
**              set pointer to next element
**         else
**                   print spaces and increment domain value by domain
**                        spacing
**              until domain value >= generation number
**              if domain value = generation number
**                   print element marker
**                   increment domain value by domain spacing
**                   set pointer to next element
**              else
**                   set pointer to next element
*/

     while ( temp != NULL ) {

          if   ( f > (float) temp->gen_number )
               temp = temp->next;
          else {
               for  ( ; f< (float) temp->gen_number ; f+=dpoint )
                    printf( " " );

               if   ( fabs( f - (float) temp->gen_number ) < 1.0 ) {
                    printf( "%s", temp->marker );
                    f+= dpoint;
                    temp=temp->next;
               }
               else
                    temp=temp->next;
          }
     }
}

/*--------------------------#########-----------------------------*/

void DisposeOfRangeList( range_list     *rl )

/*
**    frees up memory taken by the range_list pointed to by rl
*/

{
     range_list     *temp;

     while( rl != NULL ) {
          temp = rl;
          rl = rl->next;
          temp->next = NULL;
          Free( temp );
     }
}

/*--------------------------#########-----------------------------*/
```

```
void InitializeScores( void )

/*
**    initializes scores for use
*/

{
      scores.score_list_length = 0;
      scores.best_score.max = MIN_SCORE;
      scores.best_score.min = MAX_SCORE;
      scores.average_score.max = MIN_SCORE;
      scores.average_score.min = MAX_SCORE;
      scores.last_insert = time(NULL);
      scores.gen_first = NULL;
      scores.gen_last = NULL;
      scores.best_first = NULL;
      scores.avg_first = NULL;
}

/*----------------------------##########-----------------------------*/

void AddToScoreList(      int       generation_number,
                          float     best_score,
                          float     average_score )

/*
**    adds new score to scores in ascending generation number order,
**    descending best score order and descending average score order
*/

{
      score_list      *sp, *current, *previous;

/*
**    allocate space for new score list element
*/

      sp = New( score_list );

/*
**    set details of new score list element
*/

      sp->generation_number = generation_number;
      sp->best_score = best_score;
      sp->average_score = average_score;
      sp->completion_time = time( NULL ) - scores.last_insert;
      sp->gen_next = NULL;
      sp->best_next = NULL;
      sp->avg_next = NULL;

insert_into_list:
```

```
/*
**   if score list is empty insert at front
*/

     if    ( scores.gen_first == NULL ) {
           scores.gen_first = sp;
           scores.gen_last = sp;
           scores.best_first = sp;
           scores.avg_first = sp;
     }
     else {

/*
**   otherwise insert at end in generation order
*/
           scores.gen_last->gen_next = sp;
           scores.gen_last = sp;

/*
**   if best score is greatest, insert at front in best score order
*/

           if    ( best_score > scores.best_score.max ) {
                 sp->best_next = scores.best_first;
                 scores.best_first = sp;
           }
           else {

/*
**   otherwise insert in front of first lower best score
*/

                 current = scores.best_first;
                 while ( current != NULL
                 && current->best_score >= sp->best_score ) {
                       previous = current;
                       current = current->best_next;
                 }
                 sp->best_next = current;
                 previous->best_next = sp;
           }

/*
**   if average score is greatest,
**        insert at front in average score order
*/

           if    ( average_score > scores.average_score.max ) {
                 sp->avg_next = scores.avg_first;
                 scores.avg_first = sp;
           }
           else {
```

```
/*
**    otherwise insert in front of first lower average score
*/

                current = scores.avg_first;
                while ( current != NULL
                && current->average_score >= sp->average_score ) {
                     previous = current;
                     current = current->avg_next;
                }
                sp->avg_next = current;
                previous->avg_next = sp;
           }
      }

/*
**    update scores information
*/

      scores.score_list_length += 1;

      if    ( best_score > scores.best_score.max )
            scores.best_score.max = best_score;
      if    ( best_score < scores.best_score.min )
            scores.best_score.min = best_score;
      if    ( average_score > scores.average_score.max )
            scores.average_score.max = average_score;
      if    ( average_score < scores.average_score.min )
            scores.average_score.min = average_score;

      scores.last_insert = time( NULL );
}

/*--------------------------#########------------------------------*/

void PlotScores( void )

/*
**    plots graph of average (a) and best (b) scores for each
**    generation
*/

{

/*
**    determine range parameters
*/

      float     rmax = (float) ceil( scores.average_score.max * 10.0)
                      / 10.0;
      float     rmin = (float) floor( scores.best_score.min * 10.0 )
                      / 10.0;
      float     range = rmax - rmin;
      float     rstep = range / 4.0;
      float     rpoint = rstep / 5.0;
```

```
/*
**   determine domain parameters
*/

     int        dmax = ( scores.score_list_length / 51 + 1 ) * 50;
     int        dstep = dmax / 5;
     float      dpoint = (float) dstep / 10.0;

     score_list      *ap, *bp;
     int             i;
     float           f;
     char            marker[20];
     range_list      *rl;

     ap = scores.avg_first;
     bp = scores.best_first;

/*
**   display current generation information
*/

     printf( "Scores     CURRENT Gen = %4d Best = %8.6f Avg = %8.6f"
          " Lifespan %3d:%2d\n", scores.gen_last->generation_number,
          scores.gen_last->best_score,
          scores.gen_last->average_score,
          (int) ( scores.gen_last->completion_time / 601 ),
          (int) ( scores.gen_last->completion_time % 601 ) );

/*
**   for each point in the range
**        display the scale
**        add best elements which are in range to range list
**        add average elements which are in range to range list
**        plot the range list
**        dispose of the range list
*/

     for  ( i=20 , f=rmax ; i >= 0 ; i-- , f-=rpoint ) {

          if   ( i % 5 == 0 )
               printf( "%6.3f+", f );
          else
               printf( "       -" );

          rl = NULL;

          while ( bp != NULL && bp->best_score > f ) {
               if   ( bp != scores.gen_last )
                    strcpy( marker, "b" );
               else
                    sprintf( marker, "CB (%8.6f)", bp->best_score );
               rl = AddToRangeList( rl, bp->generation_number,
                    marker);
               bp = bp->best_next;
          }
```

```
            while ( ap != NULL && ap->average_score > f ) {
                if    ( ap != scores.gen_last )
                      strcpy( marker, "a" );
                else
                      sprintf( marker, "CA (%8.6f)", ap->average_score);
                rl = AddToRangeList( rl, ap->generation_number,
                      marker );
                ap = ap->avg_next;
            }

            PlotRangeList( rl, dpoint );

            DisposeOfRangeList( rl );

            printf( "\n" );
        }

        printf( "      " );

    /*
    **    display the domain scale
    */

        for   ( i=0 ; i <= 50 ; i++ )
            if    ( i % 10 == 0 )
                  printf( "+" );
            else
                  printf( "|" );

        printf( "\n    " );

        for   ( i=0 ; i <= 50 ; i++ )
            if    ( i % 10 == 0 )
                  printf( "%4d  ", (int) ( i * dpoint ) );
            else
                  if    ( i % 2 == 0 )
                        printf( " " );

        printf( " Generation\n" );
    }

    /*--------------------------#########-----------------------------*/

    void DisposeOfScores( void )

    /*
    **    frees up memory taken by scores
    */

    {
        score_list      *sp;
```

```
        while ( scores.gen_first != NULL ) {
                sp = scores.gen_first;
                scores.gen_first = scores.gen_first->gen_next;
                sp->gen_next = NULL;
                sp->best_next = NULL;
                sp->avg_next = NULL;
                Free( sp );
        }
        InitializeScores();
}

/*----------------------------##########------------------------------*/
```

B.3 Task Files

```
/*****************************************************************
**                                                              **
**   filename:       xortask.def                                **
**   programmer:     o.diessel                                  **
**   modified:       901016ofd                                  **
**   description:    task definition file for XOR problem       **
**   remark:         copy to taskdef.h and MAKE gentest to run  **
**                   XOR problem                                **
**   see also:       xornet.def                                 **
**                                                              **
*****************************************************************/
/*
**   the network is presented with the four 2 variable binary input
**   combinations and is expected to recognize the XOR function
*/

#define   TRAIN_IN                  0
#define   TRAIN_OUT                 1

unsigned  training_set[TEST_MAX][2] = { 0,   0,
                                        1,   1,
                                        2,   1,
                                        3,   0    };
```

```
/*****************************************************************
**                                                              **
**    filename:       xornet.def                                **
**    programmer:     m.bentink                                 **
**    modified:       901016ofd                                 **
**    description:    network definition header for XOR problem **
**                    defines network parameters                **
**    remarks:        copy to netdef.h and MAKE gentest to run  **
**                    XOR problem                               **
**    see also:       xortask.def                               **
**                                                              **
*****************************************************************/

#define   NO_INPUTS       2
#define   NO_OUTPUTS      1
#define   NO_LAYERS       2
#define   NO_NEURONS      (NO_INPUTS * NO_LAYERS)

#define   TEST_MIN        0
#define   TEST_MAX        4
#define   TEST_STEP       1
#define   MAX_NO_ERRORS   ((TEST_MAX - TEST_MIN)*NO_OUTPUTS/TEST_STEP)
#define   CYCLE_MAX       10

#define   ACTIVATION_MIN 0.0005

#define   TASK_NAME       "XOR"
```

```
/*****************************************************************
**                                                              **
**   filename:       partask.def                                **
**   programmer:     o.diessel                                  **
**   modified:       901016ofd                                  **
**   description:    task definition file for parity problem    **
**   remark:         copy to taskdef.h and MAKE gentest to run  **
**                   parity problem                             **
**   see also:       parnet.def                                 **
**                                                              **
*****************************************************************/
/*
**   the network is presented with bit strings 7 bits long and is
**   expected to determine whether the string has even parity
**   (output = 1) or odd parity (output = 0)
*/

#define   TRAIN_IN                  0
#define   TRAIN_OUT                 1

unsigned  training_set[TEST_MAX][2] = { 0,   1,
                                        32,  0,
                                        2,   0,
                                        80,  1,
                                        5,   1,
                                        40,  1,
                                        10,  1,
                                        84,  0,
                                        21,  0,
                                        127, 0    };
```

```
/*****************************************************************
**                                                              **
**   filename:      parnet.def                                  **
**   programmer:    o.diessel                                   **
**   modified:      901016ofd                                   **
**   description:   network definition header for parity problem **
**                  defines network parameters                  **
**   remarks:       copy to netdef.h and make gentest to run     **
**                  parity problem                              **
**   see also:      partask.def                                 **
**                                                              **
*****************************************************************/

#define   NO_INPUTS       7
#define   NO_OUTPUTS      1
#define   NO_LAYERS       3
#define   NO_NEURONS      (NO_INPUTS * NO_LAYERS)

#define   TEST_MIN        0
#define   TEST_MAX        10
#define   TEST_STEP       1
#define   MAX_NO_ERRORS   ((TEST_MAX - TEST_MIN)*NO_OUTPUTS/TEST_STEP)
#define   CYCLE_MAX       10

#define   ACTIVATION_MIN  0.0005

#define   TASK_NAME       "PARITY"
```

B.4 Make Files

```
/*****************************************************************
**                                                              **
**   filename:      gentest.prj                                 **
**   programmer:    o.diessel                                   **
**   modified:      901021ofd                                   **
**   description:   TurboC project file                         **
**   comment:       Define the appropriate dos PATHNAME,        **
**                  define the appropriate netdef.h & taskdef.h **
**                  files, or copy ???net.def to netdef.h and   **
**                  ???task.def to taskdef.h, and delete this   **
**                  header to MAKE gentest.exe using Borland's  **
**                  TurboC                                      **
**                                                              **
*****************************************************************/
\PATHNAME\gentest
\PATHNAME\genetic
\PATHNAME\scrnplot
\PATHNAME\memory
\PATHNAME\random
```

```
/*****************************************************************
**                                                              **
**   filename:      makexor                                     **
**   programmer:    o.diessel                                   **
**   modified:      901027ofd                                   **
**   description:   use on Apollo to make XOR program xortest   **
**   see also:      modify #include & #define statements in     **
**                                                              **
**                        gentest.c                             **
**                        genetic.c                             **
**                        random.c                              **
**                        scrnplot.c                            **
**                        random.c                              **
**                                                              **
*****************************************************************/
OBJS = xortest.o xorgen.o scrnplot.o memory.o random.o

xortest:  $(OBJS)
              /bin/cc $(OBJS) -o xortest

xortest.o:    netdef.h genetic.h extendc.h random.h scrnplot.h
              cp xornet.def netdef.h
              /bin/cc -O -c gentest.c -o xortest.o

xorgen.o: memory.h netdef.h genetic.h extendc.h random.h taskdef.h
              cp xornet.def netdef.h
              cp xortask.def taskdef.h
              /bin/cc -O -c genetic.c -o xorgen.o

scrnplot.o: memory.h scrnplot.h
```

```
/*****************************************************************
**                                                              **
**   filename:      makepar                                     **
**   programmer:    o.diessel                                   **
**   modified:      901027ofd                                   **
**   description:   use on Apollo to make parity program partest **
**   see also:      modify #include & #define statements in     **
**                                                              **
**                        gentest.c                             **
**                        genetic.c                             **
**                        random.c                              **
**                        scrnplot.c                            **
**                        random.c                              **
**                                                              **
*****************************************************************/
OBJS = partest.o pargen.o scrnplot.o memory.o random.o

partest:  $(OBJS)
               /bin/cc $(OBJS) -o partest

partest.o:     netdef.h genetic.h extendc.h random.h scrnplot.h
               cp parnet.def netdef.h
               /bin/cc -O -c gentest.c -o partest.o

pargen.o: memory.h netdef.h genetic.h extendc.h random.h taskdef.h
               cp parnet.def netdef.h
               cp partask.def taskdef.h
               /bin/cc -O -c genetic.c -o pargen.o

scrnplot.o: memory.h scrnplot.h
```

Bibliography

[1] Bentink M., "Neural Networks, A genetic Breeding Algorithm", Fourth Year Bachelor of Engineering Project, University of Newcastle, (1989)

[2] Penfold H.B., Diessel O.F., and Bentink M.W., "A genetic breeding algorithm which exhibits self-organizing in neural networks", Presented at IASTED International Symposium Artificial Intelligence and Neural Networks, Zürich, Switzerland, June 1990 (proceedings to appear)

[3] Kauffman S. and Levin S., "Towards a General Theory of Adaptive Walks on Rugged Landscapes", J. theor. Biol, Vol. 128, pp 11-45 (1987)

[4] Baba N., "A new approach to finding the global minimum of error functions in neural networks", Neural Networks, Vol. 2, pp 367-373, (1989)

[5] Bartlett P., and Downs D., "Training a neural network with a genetic algorithm", First Australian Conference on Neural Networks, Sydney, January 1990

[6] Rumelhart D.E., Hinton G.E. and Williams R.J., "Learning internal representations by error propagation", in Parallel Distributed Processing, Vol. 1, Eds. Rumelharhart D.E. and McClelland J.L., MIT Press, Cambridge, MA, pp 318-362, (1986)

[7] Lippmann R.P., "An introduction to computing with neural nets", IEEE ASSP Magazine, pp. 4-22 (1987)

[8] Vemuri V., "Artificial neural networks: an introduction", Neural Networks, Artificial Neural Networks: Theoretical Concepts, Ed. Vemuri V., IEEE Computer Society Press Technology Series, pp 1-11 (1988)

[9] Lapedes A. and Farber R., "How neural nets work", American Institute of Physics Press

[10] Palmer R., "Neural Nets", Complex Systems, SFI Studies in the Sciences of Complexity, Ed. Stein D., Addison-Wesley, Reading, Ma. pp. 439-461 (1989)

[11] Walbridge C., "Genetic algorithms - the calculated solution", The Australian, pp 48,49,56, February 21, 1989

[12] Holland J.H., "Genetic algorithms and classifier systems: foundations and future directions", Genetic Algorithms and their Applications, Proceedings of the Second International Conference on Genetic Algorithms, Ed. Grefenstette J.J., Lawrence Erlbaum, Hillsdale, N.J., pp. 82-89 (1987)

[13] Cohoon J.P., Hegde S.U., Martin W.N., and Richards D., "Punctuated Equilibria: a parallel genetic algorithm", Genetic Algorithms and their Applications, Proceedings of the Second International Conference on Genetic Algorithms, Ed. Grefenstette J.J., Lawrence Erlbaum, Hillsdale, N.J., pp. 148-154 (1987)

[14] Anthony D., Hines E., Barham J., and Taylor D., "The use of genetic algorithms to learn the most appropriate inputs to a neural network", (unpublished article)

[15] Goldberg D.E., and Richardson J., "Genetic algorithms with sharing for multimodal function optimization", Genetic Algorithms and their Applications, Proceedings of the Second International Conference on Genetic Algorithms, Ed. Grefenstette J.J., Lawrence Erlbaum, Hillsdale, N.J., pp. 41-49 (1987)

[16] Whitley D., "Using reproductive evaluation to improve genetic search and heuristic discovery", Genetic Algorithms and their Applications, Proceedings of the Second International Conference on Genetic Algorithms, Ed. Grefenstette J.J., Lawrence Erlbaum, Hillsdale, N.J., pp. 108-115 (1987)

[17] Pineda F.J., "Generalization of back-propagation to recurrent neural networks", Physical Review Letters, Vol. 59, pp. 2229-2232 (1987)