Chapter 1

Genetic Synthesis of Recurrent Neural Networks

1.1 Introduction

Many of the systems we wish to model in the real world are often nonlinear dynamical systems. This is particularly true in the controls area in which we wish to model the forward or inverse dynamics of systems such as airplanes, engines, rockets, spacecrafts, and robots. Many of the systems also tend to exhibit state dependent behavior. The problem of developing a system for context dependent pattern recognition, in which the classification of a pattern is dependent on the context in which it appears, is an example.

When applying neural networks to problems involving nonlinear dynamical or state dependent systems, neural networks with feedbacks can in some cases provide significant advantages over purely feedforward networks. The feedbacks allow for recursive computation and the ability to represent state information. In some cases a system with feedbacks is equivalent to a much larger and possibly infinite feedforward system. Neural networks employing an architecture that incorporates feedbacks are referred to as Recurrent Neural Networks (RNNs).

Since the rebirth of artificial neural networks in the mid 1980s, a significant amount of work has been done in studying the capabilities and limitations of RNNs, and in applying them to various problems of pattern recognition and control. However, the use of RNNs has not been nearly as extensive as that of feedforward networks. The primary reason for this stems from the difficulty of developing generally applicable learning algorithms for RNNs. The feedforward networks benefit from the fact that they can be trained using gradient decent optimization algorithms such as the backpropagation algorithm [Rumelhart and McClelland, 1986]. Unfortunately, since the gradient of the error with respect to the connection strength is not easily solvable in general for RNNs such gradient based optimization algorithms are not always applicable.

In this thesis we explore the applicability of a heuristic optimization algorithm, referred to as the Genetic Algorithm (GA), to the problem of finding the network parameters and possibly even the network architecture for a RNN. The GA has proven to be quite successful on a number of other difficult optimization problems [Goldberg, 1989] and has even been applied to the synthesis (determining both the network parameters and architecture) of feedforward neural networks [Whitley, Starkweather, and Bogart, 1990]. However, the use of GAs for synthesizing RNNs still needs to be extensively explored.

1.2 Background

Much of the earlier research in developing training algorithms for RNNs has been focused on using gradient descent algorithms. There are basically two classes of gradient descent algorithms used with RNNs: backpropagation through time and recursive backpropagation, also referred to as dynamic propagation or real time recurrent learning. Although both algorithms are based on variations of the backpropagation algorithm, they tend to be much more complicated then the backpropagation algorithm used for feedforward networks. Also, both algorithms have a different formulation depending on weather the network to be trained is a discrete time or a continuous time RNN.

The backpropagation through time algorithm is based on converting the network from a feedback system to a purely feedforward system by unfolding the network over time. Thus, if the network is to process a signal that is n time steps long, then n copies of the network are created and the feedback connections are modified so that they are feedforward connections from one network to the subsequent network. The network can then be trained as if it is one large feedforward network with the modified weights being treated as shared weights. The application of this algorithm to discrete time RNNs was popularized by its description and use in the well-known PDP book [Rumelhart and McClelland, 1986]. The continuous time version of backpropagation through time was later derived by Pearlmutter

[Pearlmutter, 1989]. A major limitation of the backpropagation through time algorithm is that one must know in advance how many copies of the network to create when unfolding the network over time.

The recursive backpropagation algorithm is based on recursively updating the derivatives of the output and error. These updates are computed using a sequence of calculations at each iteration. The weights are updated either after each iteration or after the final iteration of the epoch. This algorithm was proposed for discrete time RNNs by a number of different researchers [Kuhn, 1987; Mozer, 1988; Robinson and Fallside, 1987; Williams and Zipser, 1989]. The continuous time version of recursive backpropagation was first proposed by Pineda [Pineda, 1988]. The major disadvantage of this algorithm is that it requires an extensive amount of computation at each iteration.

Another algorithm which has been shown to descend an error function and has been applied to RNNs is the deterministic Boltsman Machine learning rule [Peterson and Anderson, 1987]. Unlike backpropagation based learning which requires a different formulation for discrete and continuous time networks the Boltsman Machine learning rule can be applied equally well to both types of RNNs. The Boltsman Machine learning rule treats the system error of the RNN as an energy function that must be minimized. A generating function is used to generate the next potential solution based on the current solution. An acceptance function is used to decide weather to keep the current solution or to accept the newly generated solution based on the difference in the energy of the two solutions. An annealing schedule is used to reduce the neighborhood of the newly generated solutions and also to increase the probability of accepting better solutions and rejecting worse solutions.

Simulated annealing, which works similar to Boltsman Machine learning, but uses a different generating function and annealing schedule, has also been applied to training RNNs [Van den Bout and Miller, 1989]. If an appropriate annealing schedule is used, both of these methods are guaranteed to find the global minimum. However, the amount of time required to find the global minimum may be unacceptable for difficult problems. Both algorithms are also very sequential and do not lend easily to parallel implementation.

The use of evolutionary based algorithms for training neural networks has recently begun to receive a considerable amount of attention. Much of the research however has focused on the training of feedforward networks [Fogel, Fogel, and Porto, 1990; Whitley, Starkweather, and Bogart, 1990]. Research on applying evolutionary algorithms to RNNs is currently quite limited, but seems to be increasing. While work on this thesis was in progress, some papers have appeared in the neural network literature dealing strictly with an evolutionary approach to synthesizing RNNs [Angeline, Gregory, and Jordan, 1994; Beer and Gallagher, 1992; Bornholdt and

Graudenz, 1992]. However, most of these papers describe approachs which use ad hoc operators for combining and mutating networks and do not use the much studied bit string representation for encoding and manipulating the networks.

One of the difficulties in using a bit string representation is that it does not allow an easy way to encode a variable number of parameter. When encoding neural networks the number of parameter will vary depending on the architecture of the network. Thus, many researchers have avoided using a bit string representation when the architecture of the network is also supposed to be determined by the evolutionary process. Later in this thesis we present and examine a modification to the standard genetic algorithm which allows the network architecture to be selected while still using a bit string representation and the usual genetic operators.

1.3 Network Architecture

The recurrent neural network architecture used throughout this thesis is a two layer network in which the second layer is very similar to the Hopfield network. This network is shown in Figure 1.

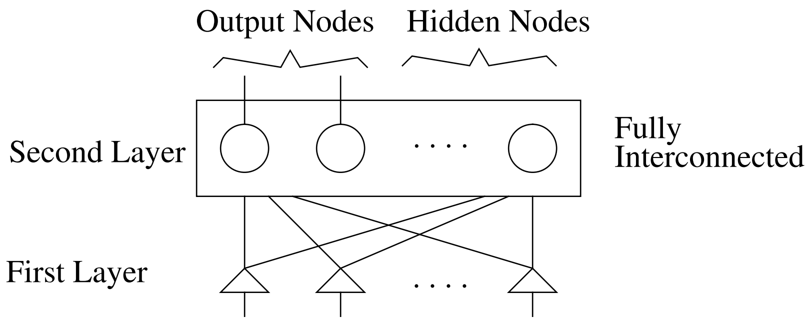


Figure 1: A two layer network architecture is used. The first layer consists of input nodes which range compress the input and fan it out to all the nodes in the second layer. The nodes in the second layer are fully interconnected and each has an internal state. Some of the nodes in the second layer are output nodes while the remaining are hidden nodes.

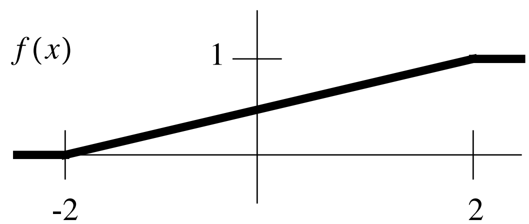


Figure 2: The activation function used for the nodes in the second layer is a piecewise linear function with hard saturation limits at 2 and -2.

The first layer is composed of input nodes which simply range compress the applied input (based on prespecified range limits) so that it is in the range [0, 1] and fanout the result to all nodes in the second layer. The nodes in the second layer have an internal state and compute their outputs synchronously based on this state. The outputs are computed using the piecewise linear activation function shown in Figure 2. A subset of the nodes in this layer are taken to be "output nodes" and the remaining serve as "hidden nodes". The dynamics of the network are described by the following differential equations when continuous time nodes are desired:

$$\dot{x}_i(t) = -A_i x_i(t) + \sum_j I_j(t) w_{ji} + \sum_k o_k(t) w_{ki} + \theta_i$$
$$o_i(t) = f(x_i(t))$$

In a computer simulation these differential equations are implemented as:

$$o_i(t+\Delta t) = f(x_i(t+\Delta t))$$
$$x_i(t+\Delta t) = x_i(t) + \Delta x_i(t)\Delta t$$
$$\Delta x_i(t) = -A_i x_i(t) + net_i(t)$$
$$net_i(t) = \sum_j I_j(t) w_{ji} + \sum_k o_k(t) w_{ki} + \theta_i$$

The first order Euler equation is used to compute the next state of the network from the current state and inputs. $f(x)$ is the nonlinear activation function described earlier. o_i is the output of node i in the second layer and x_i is its internal state. $I_j(t)$ is the input to the jth node in the second layer at time t. w_{ji} is the weight from the jth node in the first layer to the ith node in the second layer. w_{ki} is the weight from the kth node in the second layer to the ith node in the second layer. θ_i is the threshold parameter of the ith node in the second layer and A_i is its nonnegative time constant. For all experiments described in this thesis the range of the parameters w_{ji}, w_{ki} and θ_i were limited between [-32, 32] and the parameter A_i was limited between [0, 32].

When discrete time nodes are required the dynamics are defined by the following equations:

$$o_i(t+1) = f(x_i(t+1))$$
$$x_i(t+1) = net_i(t)$$
$$net_i(t) = \sum_j I_j(t) w_{ji} + \sum_k o_k(t) w_{ki} + \theta_i$$

This formulation is reached by setting A_i and Δt equal to 1 in the continuous time equations.

1.4 Goals of this Research

The main goal of this research is to investigate the genetic algorithm as a means for finding the network parameters and architecture that allow a RNN to solve the given problem. Our secondary goal is to gain experience in apply GAs to RNN. The GA has many parameters and methods that can be varied in an attempt to improve the performance of the GA. A better understanding of the relationship between these variables and their effect on the final performance of the GA is required so that a correct assessment of the GAs capabilities can be made.

1.5 Overview of the Thesis

We have described in this chapter the architecture of the RNN that will be used in this thesis. We have also given a background of the previous work that has been done with the application of various training algorithms to

RNNs. In the next chapter we describe the standard genetic algorithm. In Chapter three we apply this algorithm to the XOR problem. Chapter four examines possible ways of improving the standard genetic algorithm using the results of Chapter three as a basis for comparison. Chapter five and six apply the best GA found in Chapter four to problems which are more dynamical. In Chapter five we try to find RNNs that learn finite state machines from examples. In Chapter six we apply RNNs to the problem of balancing an inverted pendulum. Finally in Chapter seven we discuss the lessons learned in applying RNNs to dynamical problems and the applicability of GAs as a heuristic optimization algorithm for finding the network parameters and architecture. We also discuss some possible directions for future research.

Chapter 2

The Standard Genetic Algorithm

2.1 Introduction

The problem of finding a set of parameters for a neural network which allows it to solve the given problem can be viewed as a parameter optimization problem. The range of the various parameters such as weights, thresholds and time constants can be bounded between a minimum and maximum value so that the size of the search space is finite. Once the problem is viewed in this way, the various methods available for solving multivariable function optimization problems become applicable. However, methods which make use of gradient information are not applicable in our case since the function we are trying to optimize may not be differentiable. Of particular interest are methods which use only the value of the function in the course of the optimization process. Simulated annealing and genetic algorithms are two optimization methods which use only the value of the function in trying to find the global optimum.

In the method of simulated annealing a current best solution is maintained and the next solution is generated from the current solution. The distance of the generated solution from the current solution can be described by a Gaussian distribution. The generated solution is accepted as the current best solution using a probability computed by applying a sigmoid function to the difference between the function value of the generated and current solution. Both the distribution of the generated solutions and the probability of their acceptance are a function of an annealing schedule such that as time progresses the average distance of the generated solutions from the current solution gets shorter and the probability of accepting better solutions and rejecting worse solutions gets higher. Although this method is guaranteed to find the optimal solution it can require a prohibitively long time to do so for large problems. Also due to its sequential nature the algorithm does not lend itself to parallel implementations.

The genetic algorithm on the other hand is not guaranteed to find the optimal solution, but is capable of finding good solutions quickly [Goldberg, 1989]. Also since a population of solutions are maintained the genetic algorithm is inherently parallel.

2.2 Natural and Artificial Evolution

As described initially by Darwin, evolution is the process by which a population of organisms gradually adapt over time to better survive and reproduce in the conditions imposed by their surrounding environment. In natural evolution members of a population vary in form, function, and behavior. Much of this variation is heritable from one generation to the next. Some forms of heritable traits are more adaptive to the environment than others; that is they improve chances of surviving and reproducing. As s result, those traits become more common in the population and thus the population becomes better adapted to the environment.

Natural evolution can also be viewed as an optimization problem where the objective is to find a set of traits that maximize the chances of an organism to survive and reproduce in the given environment. This view of natural evolution allows us to select the essential features of the evolutionary process so that it can be artificially applied to any optimization problem.

In artificial evolution the members of the population represent possible solutions to a particular optimization problem. The problem itself represents the environment. We must apply each solution to the problem and assign it a fitness value indicating its performance on the problem. The two essential features of natural evolution which we need to maintain are propagation of more adaptive traits to future generations and the heritability of traits from parent to offspring. By applying a selective pressure which gives better solutions a greater opportunity to reproduce we can satisfy the first criteria. To satisfy the second criteria we need to ensure that the process of reproduction preserves most of the traits of the parent solution and yet allows for diversity so that new traits can be explored.

2.3 The Genetic Algorithm

The use of artificial evolution as a heuristic search algorithm for solving difficult problems has been explored since the mid 1960's [Fogel, 1966]. Since then many approaches of implementing artificial evolution have emerged with genetic algorithms being one of them. This approach was introduced by John Holland and his associates [Holland, 1975].

The primary feature of the genetic algorithm which distinguishes it from other evolutionary algorithms is that it represents the specimen in the population as bit strings. This is analogues to the DNA strands used in nature to encode the traits of real organisms. The encoding allows the genetic algorithm to use a set of genetic operators to manipulate the bit strings when creating new specimen. These operators are similar to the types of operations that are naturally encountered by the DNA strands in real organisms during reproduction. The advantages of this approach lies in its generality [Goldberg, 1989]. The bit string representation can be used to encode a wide range of problems, not only those which can be represented as a set of parameters. Also the use of genetic operators to create new specimen allows the search algorithm to be completely independent of how the specimen are decoded and the problem to which they are applied.

In genetic algorithms each specimen in the population consists of a fixed length bit string. This is called the genotypic representation of the specimen. To represent a possible solution to a parameter optimization problem as a bit string we simply need to limit the resolution of each parameter to a finite number of bits and concatenate the binary encoded parameters to form the bit string. Before the specimen can be evaluated we need to decode the bit string to get back a set of parameters. This decoded version is called the phenotypic representation of the specimen.

The genetic algorithm starts with an initial population of randomly created bit strings. These initial specimen are decoded and applied to the problem. The result of the specimens performance on the problem is represented by a value referred to as the fitness of the specimen. A new generation of specimen are created from the current population by applying genetic operators. Not all specimen in the current population are selected equally to be parents of future generations. Rather specimen with better fitness values are selected as parents more often. This ensures that the more adaptive traits have a higher probability of being propagated to future generations. Once the new generation of specimen are created they are evaluated to determine their fitness. The current population is replaced by the new specimens which can now serve as parents in creating the next generation of specimen. The algorithm proceeds as such creating one generation after another until the average fitness of the population or the fitness of the best specimen in the population reaches a desired fitness.

2.4 Genetic Operators

There are two basic genetic operators used to create new specimen from current ones. The first is called mutation. It simply involves making a copy of the parent specimen and randomly toggling some of the bits. The probability of toggling each bit is determined by a constant called the mutation rate. The other genetic operator is called crossover. Crossover basically consists of combining the bit strings of two specimens to create a child with a new bit string. There are various ways of implementing the crossover operator. A commonly used form of crossover called 1-point crossover cuts each of the parent bit string into two substrings. Both parent strings are cut at the same randomly chosen location and the substrings after this location are swapped to create two new specimen. In the example below the parents are cut after bit location 4 and swapped to create the child specimens.

1-Point Crossover

Parent 1: XXXXXXX	Child 1: XXXXYYY
Parent 2: YYYYYYY	Child 2: YYYYXXX

Another form of crossover is called uniform crossover. This method creates a child specimen by using a probability of 50% that any given bit in the child will come from the first parent (otherwise the bit comes from the second parent). A variation of this method called the parameterized uniform crossover allows a parameter to control the probability of any given bit in the child coming from the first parent. Setting this parameter to 1 results in the child being identical to the first parent and setting it to 0 results in the child being identical to the second parent. The parameter can be set to any value in between to get the desired mix of parents. In the work described in this thesis the parameterized uniform crossover operator was used. This form of crossover was chosen because of its less disruptive and greater exploratory features [Spears and De Jong, 1990].

In genetic algorithms the crossover operator is the primary method for exploiting the diversity of the current population to find which bit combinations lead to better fitness. Thus, each new specimen is created using crossover. The drawback is that the crossover operator tends to decrease the diversity of the population rather quickly and leads to a population in which all the specimen are identical at which point the crossover operator is no longer able to produce new specimen. Likewise, if a particular bit in all the specimens had the same value there would be no way to create a specimen that had a different value for this bit if only crossover was used. In order to escape from such situations and keep the population diverse the mutation operator must be used. Thus, after a new specimen is created using crossover the mutation operator is applied. However, mutation has the negative effect of disrupting the good bit combinations (or schemas) already found. In order to minimize this disruptive effect the mutation rate is usually set to a very small number such as 0.001. The population size plays an interacting role with mutation. As the size of the population gets smaller the probability that a particular bit in all the specimen will be stuck at the same value increases. Thus, smaller populations require a relatively larger mutation rate to keep the population diverse.

2.5 Selective Pressure

The ability of the genetic algorithm to produce progressively better specimen lies in the selective pressure it applies to the population. The selective pressure can be applied in two way. One way is to create more child specimen then are maintained in the population and select only the best ones for the next generation. Even if the parents were picked randomly this method would still continue to produce progressively better specimen due to the selective pressure being applied on the child specimen. The other way to apply selective pressure is to chose better parents when creating the offsprings. With this method only as many child specimen as maintained in the population need to be created for the next generation. Even when selective pressure is not applied to the offsprings this method will continue to produce progressively better specimen, due solely to the selective pressure being applied to the parents.

Although in natural evolution both types of selective pressure seem to be at work, algorithms based on artificial evolution tend to favor the latter method of applying selective pressure. This is primarily due to the fact that the evaluation of child specimens to determine their fitness is usually the most computationally intense part of the algorithm. Thus, it is preferable to be selective about which parents are allowed to breed so that only as many specimen as maintained in the population are produced and evaluated.

One way to allow the better fit specimen in the population to reproduce at a higher rate is to use a selection method based on the roulette wheel selection technique. Each specimen in the population is represented by a slot in a roulette wheel with the size of the slot being proportional to the fitness of the specimen. Such a roulette wheel is spun to randomly select the parents with a probability proportional to their fitness.

Another method called tournament selection can also be used to apply selective pressure to the parents. This method picks N random specimens from the population and chooses from them the specimen with the best fitness. With this method one can chose the value of N to control the amount of selective pressure. If N is 1 then this method is equivalent to random selection. Increasing N causes the amount of selective pressure being applied to also increase.

In our initial experiments the roulette wheel selection method was used. However, after later comparison of the two methods (see Chapter 4) tournament selection was chosen for the remaining experiments.

2.6 Population Size and Mutation Rate

As stated earlier the crossover operator is the primary source of the genetic algorithm for creating new and unique child specimen. However, the ability of the crossover operator to produce unique specimen diminishes as the diversity of the population decreases. Thus, the ability to maintain the diversity of the population using mutation is a critical issue in genetic algorithms. The population size and mutation rate are key factors that effect the rate at which diversity decreases. Selective pressure also contributes to decreasing the diversity of the population. However, even in the absence of selective pressure there is a natural tendency for the diversity of the population to decrease. This phenomena is referred to in population genetics as genetic drift. Appendix A explores the relationship between population size, mutation rate and selective pressure on a populations diversity and tries to model these relationships analytically.

In most of the GA experiments described in this thesis we have chosen to use a relatively small initial population size of 30 specimen (or 150 specimen when a mixed size population is used), since smaller populations are generally able to converge to a solution more quickly. We have chosen to use a mutation rate of 0.01 for this population size.

2.7 Mixed Specimen Populations

Ultimately, we would like to use the GA to find not only the network parameters of a RNN, but also the network architecture. However, the difficulty in doing this is that allowing the GA to modify the network architecture requires the network architecture to be encoded by the bit strings in such a way that the existing genetic operators can be used. Otherwise, special genetic operators just for manipulating the network architecture must be introduced. Also, since a variable architecture means a variable number of network parameters and consequently variable length bit strings, the existing genetic operators would need to be modified to handle these variable length bit strings. Both approaches would require a major deviation from the standard GA described thus far.

However, a third approach can be used which does not require any changes to the GA other than a minor change to the parent selection method. This approach simply starts with a population where the specimen are networks of varying architectures and implements the parent selection method so that only compatible specimen can be crossed. The initial population would contain an equal number of specimen that had a specific number of nodes in the second layer. The first parent is chosen from the entire population using the a parent selection method such as the roulette wheel. The second parent is also chosen using the same parent selection method. However, it is chosen in such a way that it will have the same number of nodes in the second layer as the first parent. This can be accomplished by either reselecting the second parent until a compatible one is found or by limiting the selection to only the subpopulation that is compatible with the first parent.

This approach extends the selective pressure already being applied in finding the network parameters so that it is also applied in selecting a network architecture. This method of parent selection introduces direct competition between different sized networks for a presence in the population. The network size which is more quickly able to find better solutions than the competition will eventually dominate the population. Once the population is dominated by a specific network architecture the parent selection method will become equivalent to that used in a fixed size population.

This small change in the parent selection scheme will allow the standard genetic algorithm and the bit string representation of the specimens to be used while still allowing the network architecture to be selected by the evolutionary process.

Chapter 3

The XOR Problem

3.1 Introduction

To make an initial assessment of how well the standard genetic algorithm performs on the task of finding recurrent neural networks to solve a given problem we have chosen the 2-input XOR problem as a test case. The 2-input XOR problem is defined as finding a network which produces an output of 1 when the input is <0, 1> or <1, 0> and an output of 0 when the input is <0, 0> or <1, 1>.

The XOR problem was chosen since it is a simple, yet nontrivial problem for neural networks to solve. Also, the XOR problem has been extensively used as a benchmark for performance of neural network training algorithms. It should be noted that traditionally a recurrent neural network is not used for problems such as the XOR problem, or more generally for arbitrary input to output mapping problems. A multilayer feedforward network is sufficient to solve such problems [Hornik, Stinchcombe, and White, 1989].

The problem of finding a recurrent neural network for solving the XOR problem will be more difficult than finding a functionally equivalent multilayer feedforward network since we are searching in a larger space that is a super set of the multilayer feedforward network space. Also, most recurrent networks will have outputs that change dynamically with time, however our problem requires finding a recurrent network that not only produces the appropriate output for the given input, but also maintains a stable output that does not change with time. Thus, we are interested in investigating how such input/output mapping problems can be solved by a recurrent neural network. It will be interesting to see weather the networks found will use recurrent connections even though they are not necessary for this problem.

Finally, we are interested in using the evolutionary process to help determine not only the network parameters, but also the number of hidden nodes needed in the second layer. In the last experiment described in this chapter we evolve populations with mixed network sizes and compare the results against experiments evolving fixed network size populations.

3.2 Network Fitness

The fitness of a particular network at solving the XOR problem was determined by applying each of the four possible inputs and measuring the difference between the desired and actual output of the network. However, since the nodes in the second layer have an internal state, the state of the network when the input was applied will also effect the output of the network. To ensure that the evolved network is capable of converging to the correct output independent of the state it is in, steps must be taken during the evaluation to ensure that the network was not in the same state whenever a particular input was applied. Before evaluating the network for a particular input pattern, the state of the network is initialized randomly such that the internal state variable of each node is in the range [-2, 2]. This range is chosen based on the saturation limits of the activation function (-2 to 2) being used, so that the initial state of the network will span the full saturation range. Also on some trials the next input pattern is applied without changing the state of the network, so that it starts at the state it converged to for the previous input. This requires the order in which the input patterns are presented to be shuffled so that the same pattern is not consistently presented after any other pattern. A probability of 50% is used to decide whether to initialize the state of the network randomly or to keep it in the same state before presenting the next pattern.

Due to the recurrent nature of the neural network used, a certain amount of time must be allowed for the output of the network to settle whenever a new input is applied. Also, since we want the evolved network to converge and maintain a static output, we must measure the output over a period of time rather than just one instance. Otherwise, it would be possible to evolve seemingly good networks that achieve the desired value at just the measured time and then change the outputs after being measured. For the experiments discussed later a measuring time of 2 time units (or 40 simulation steps with dt = 0.05) was used after allowing 2 time units for the network to settle. During the measuring time the following equation was used to determine the fitness of the network for the given input pattern.

$$fitness(p) = \frac{1}{MN}\sum_{m}^{M}\sum_{n}^{N} 1 - (d_n(p) - o_n(t + m\Delta t))^2$$

The squared difference between the desired output and the actual output is used as the metric for measuring the networks fitness. However, since we want a larger fitness value to mean a better specimen (network), we subtract the squared difference from 1. This value is summed over all the output nodes (N, which in this case is 1) and repeated for a duration of M simulation steps. The result is averaged to give a fitness value between 0 and 1 for the applied pattern. The total fitness for the network is computed by performing the above calculation for each pattern in the set and the set of patterns may be applied several times. The following equation is used to measure the total fitness of the network.

$$fitness = \frac{1}{RP}\sum_{r}^{R}\sum_{p}^{P} fitness(p)$$

The total fitness is the average of the fitness calculated by applying the set of patterns (P) to the network R times. It should be noted that the total fitness calculated has some noise in it due to he fact that some stochastic decisions were made in the process of calculating the fitness for each pattern (such as the random initial state from which to start the network). Thus, the total fitness for the same network will differ from one evaluation to the next. The variance of the total fitness can be reduced by increasing the number of times (R) the set of patterns are applied to the network.

Because of the way the fitness function is defined it will produce values that are better then the specimens actual performance. For example the fitness of a specimen that on average produces an output that is 50% of the desired output will have a fitness of 1 - $(0.5)^2$ which is 0.75. Likewise, a specimen that has a fitness of 0.99 is really producing an average output that is 90% of the desired output, since 1 - $(0.9)^2$ = 0.99. This should be considered when choosing the desired level of fitness we want the networks to reach.

3.3 Experiments and Results

The first set of experiments were performed to measure the amount of noise in the fitness function and to get a better understanding of how much the noise was reduced by increasing the number of times the set of input patterns were applied to the network. A specimen with 3 nodes in the second layer was evaluated 20 times by presenting the pattern set only once on each evaluation. The standard deviation in the fitness for the 20 evaluations was computed. The number of times the pattern set is presented was increased to

2 and the same specimen is evaluated 20 more times and the standard deviation in fitness computed. This procedure is repeated up to 10 presentations of the pattern set on each evaluation. This procedure was repeated for 10 specimens and the resulting standard deviation curves averaged to produce the final result shown below.

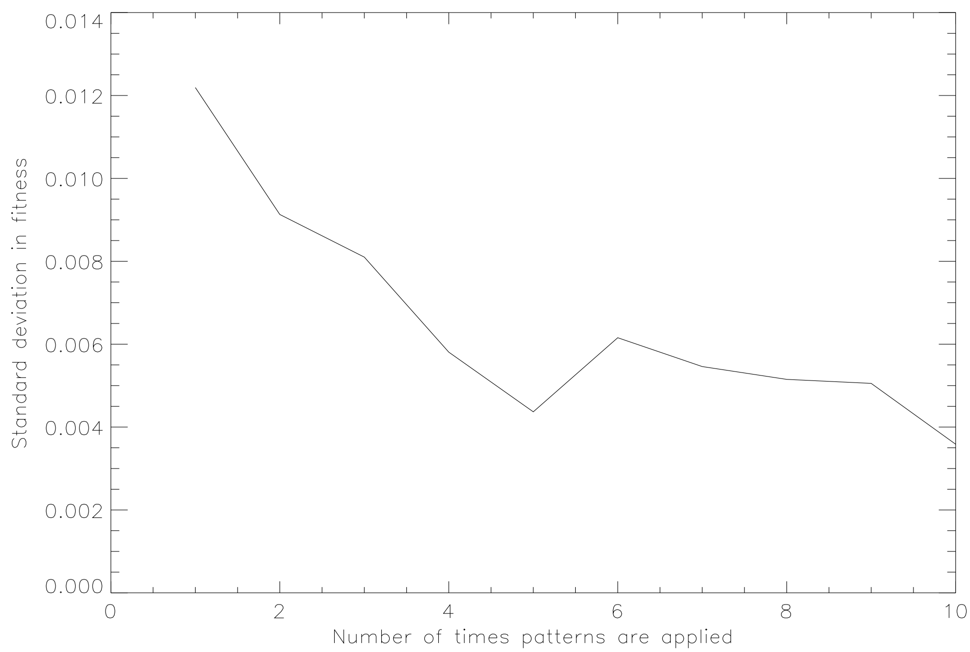


Figure 3: The amount of deviation in the fitness due to noise decreases as the number of times the pattern set is applied increases.

From this result it is apparent that not much reduction in the standard deviation is gained after about 5 presentations of the pattern set. Since increasing the number of times the pattern set is applied directly increases the computation time for fitness evaluation it is desirable to keep it as low as possible. In all subsequent experiments the input pattern set was presented 5 times in determining the specimen's fitness.

The next set of experiments were used to determine what value to use for the parameterized uniform crossover operator. Networks having 2 nodes in the second layer were evolved to solve the XOR problem. The table below shows the average number of generations that were required to evolve a network for the various values of the uniform crossover parameter. The average was taken by running each parameter value 5 times. Only the trials in which the genetic algorithm converged to a solution were used in competing the average. A solution was considered to have been found when the fitness of the best specimen exceeded 0.99. Although this may seem to be an overly strict criteria it really corresponds to the average network output being 90% of the desired output because of the way we have defined the fitness measure. If a solution was not found within 500 generations the search was terminated and the genetic algorithm is considered not to have converged to a solution.

The number of specimens in the population on all runs was 30. The mutation rate was fixed at 0.01. The networks were given 2 time units to settle and were then measured for 2 time units. The parameter for uniform crossover was varied between 0.7, 0.8 and 0.9. The selected parents were ordered based on fitness (so that the parent with higher fitness was the first parent) before applying the crossover operator. Thus, the bias due to the crossover parameter was always in favor of the better fit parent. From the results of these experiments a value of 0.8 was chosen for the parameterized uniform crossover operator for all subsequent experiments.

Table 1: Effects of uniform crossover bias on the performance of the GA.

Parameter for uniform crossover	Convergence ratio	Average number of generations
0.7	4:5	352
0.8	3:5	134
0.9	3:5	172

In the next set of experiments recurrent neural networks of varying number of nodes in the second layer were evolved to solve the XOR problem. Networks having 2, 3, 4, and 5 nodes in the second layer were evolved (although only one node is used for output; the rest are hidden nodes). The table below shows the average number of generations that were required to evolve a network for each of the various number of nodes in the second layer. The average was taken by running each network size 50 times. Only the trials in which the genetic algorithm converged to a solution were used in computing the average. The criteria for stopping and the parameters for the genetic algorithm were the same as in the previous set of experiments.

Table 2: Performance of the GA for various number of nodes in the second layer.

Number of nodes in second layer	Convergence ratio	Average number of generations
2	30:50	156
3	35:50	176
4	47:50	194
5	46:50	229

Although solutions can be found for networks with only two neurons in the second layer, these networks tend to get stuck in local minimums quite easily and only find solutions about 60% of the time. Using a larger population size, or a slightly larger mutation rate should help improve this ratio. However, even with a relatively small population the other network sizes are able to find solutions about 90% of the time.

It is interesting to note that when a RNN is used, it is possible to find solutions for the XOR problem with networks that have only one hidden node (two nodes in the second layer). Feedforward networks require at least two hidden nodes to be able to solve the XOR problem.

In comparing the average number of generations needed for finding a solution, it should be noted that the number of network parameters that need to be determined increases as the square of the number of nodes in the second layer. The total number of free parameters for networks with 2, 3, 4 and 5 nodes in the second layer are 12, 21, 32 and 45 respectively. From the experimental data it appears that the number of generations needed to find a solution does not seem to be increasing quadratically with increases in the network size. Rather the rate of increase seems to be linearly proportional to the number of nodes in the network.

The following figure shows a typical path towards a solution as the genetic algorithm tries to find a network with 3 nodes in the output layer to solve the XOR problem.

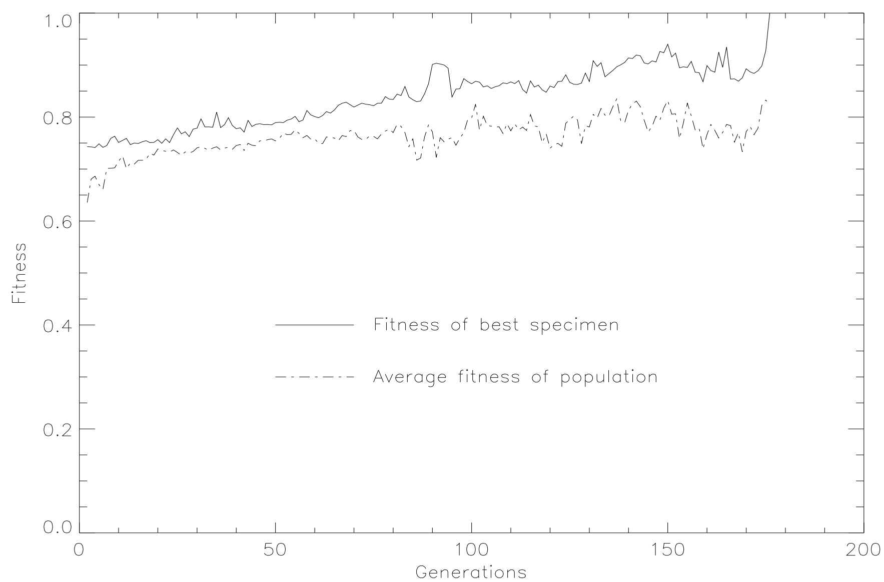


Figure 4: The rate of increase in fitness of the best specimen and the average of the population for a typical run.

The solid line shows the fitness of the best specimen and the doted line shows the average fitness of the population. From this graph it is apparent that specimens which are much more fit then the average specimen appear through out the search, but are lost just as quickly, since the parent specimens are discarded once the child specimens are created. However, through out the run there is a gradual tendency for both the average fitness and the fitness of the best specimen to improve.

By examining the interconnection weights of the nodes in the second layer it was found that in all observed cases the networks found used recurrent connections and even self feedback connections. This was true regardless of the number of nodes in the second layer. The solutions found by the GA are not at all the architectural equivalent of feedforward networks. It seems that given the ability to use recurrent and feedback connections it is easier for the GA to find solutions that make use of these connections than solutions which do not use them.

In the final set of experiments the genetic algorithm was used to find not only the network parameters, but also the number of neurons to use in the network. In these experiments the specimen in the population had a varying number of nodes in the second layer. The number of nodes in the second layer were limited to between 1 and 5. A population size of 150 specimens with 30 specimens for each of the different second layer size was used. Specimen with different number of neurons in the second layer require different number of parameters to encode them and thus have bit strings of different lengths. Since the crossover operator requires the two parents to have bit strings that are of the same length, the two parents selected must have the same number of neurons in the second layer. The experiments were run exactly like the pervious set of experiments, except that the method for selecting the second parent was slightly different. If the second parent selected by the roulette wheel did not have the same number of neurons in the second layer as the first parent then another attempt was made. If a second parent with a matching number of neurons in the second layer was not found in 10 attempts, then the first parent was also used as the second parent. The following table shows the results of 100 trials of these experiments.

Table 3: Performance of the GA when a mixed size population is used.

Number of nodes in second layer	Number of times dominated population	Number of times converged	Average number of generations
1	8	0	NA
2	34	19	117
3	33	33	127
4	14	14	118
5	11	11	127

It was found that in these experiments one of the five network types would dominate the population and cause the other network types to go extinct. How often each of the five network sizes dominated the population is shown in the second column of the table. The network type which can most readily improve its performance is most likely to dominate the population. Thus, the smaller sized (1-3 nodes) networks dominated the population about 75% of the time as compared to the larger sized (4 and 5 nodes) networks. However, the smaller sized networks are also more likely to get trapped in a local minimum or may not even be able to solve the problem at all. Thus, the percent of time the smaller sized networks converged is about 70% compared to 100% for the larger sized networks.

The competition between networks of different sizes causes the average number of generations to find a solution to be significantly less than what it would be if the different sized networks are evolved independently. Table 2 suggests that if we evolved in parallel 5 independent groups of networks with each group having a different number of nodes in the second layer we can at best expect to find solutions in about 156 generations on average. But using a mixed size population with the same total number of specimen allows us to find solutions in about 120 generations on average as suggested by Table 3. Also, using a mixed size population seems to improve the convergence ratio (the number of trials in which a solution is found) particularly for the larger (3-5 nodes in second layer) networks. When fixed network size populations are evolved independently the convergence ratio is about 92% when there are 3-5 nodes in the second layer, whereas for a mixed network size population the same convergence ratio is 100%. Thus, evolving with a mixed size population seems to improve both the speed of convergence to a solution and also the convergence ratio.

3.4 Discussion of Results

The experiments described in the previous section demonstrate that the genetic algorithm can be applied to finding the network parameters and architecture for recurrent neural networks. In fact the GA was able to discover RNNs that require only one hidden node to solve the XOR problem. Feedforward networks require at least two hidden nodes in order to solve this problem.

When different sized networks were evolved independently with even a relatively small populations sizes of 30 specimen the GA was able to find solutions in an average of about 188 generations. The convergence rate was about 60% for the smaller sized networks (2, or 3 nodes in the second layer) and about 90% for the larger sized networks (4, or 5 nodes in the second layer).

In all of the observed solutions it was noticed that the solutions found by the GA used recurrent and even self feedback connections. This was true regardless of the number of nodes in the second layer. Since a feedforward network can solve the XOR problem it was not necessary for the solutions to have recurrent and self feedback connections. The fact that the solutions found by the GA are not at all architecturally equivalent to feedforward networks suggests that it is easier for the GA to find solutions that use recurrent and feedback connections than solutions which do not use them.

When mixed size populations were evolved with the initial population containing 30 networks of sizes 1 to 5 in the second layer, the GA was able to find solutions in an average of about 125 generations. Also the convergence rate improved to about 69% for the smaller sized networks (1, 2, or 3 nodes in the second layer) and to 100% for the larger sized networks. This improvement can be attributed to the competition between the different sized networks for a greater presence in the population.

Chapter 4
Enhancing the Genetic Algorithm

4.1 Introduction

The genetic algorithm has many parameters that can be experimented with to possibly improve the performance of the algorithm. The most obvious are parameters such as the mutation rate and population size. In addition the GA uses various methods that can be altered. Some of these include the method to use for encoding the bit strings, method of implementing the crossover operator, and method for selecting parents.

The number of parameters and methods that can be modified in a GA is large enough that it is virtually impossible to do an exhaustive search for the best combination. However, even if we were to find a good set of GA parameters and methods we would still be left with the question of whether these are applicable in general or are just specific to the problem to which the GA was applied. For example, a parameter which caused the algorithm to be very greedy could very likely improve the performance of the GA when it is applied to a problem that does not have many local minimums in its search space. However, this same parameter would tend to produce worse results if the GA is applied to a problem with a search space cluttered with local miniums. Thus, the problem of finding truly good GA parameters is made more difficult because we must discern between general improvements in performance and improvements that are specific to the particular problem.

Fortunately, in our case we are not as much concerned about weather the parameters and methods improves performance in general as long as it shows a significant improvement for our particular problem of finding RNN parameters. Our results of testing certain parameter changes will only serve as single data points in the more general search for good GA parameters.

In this chapter we explore three possible methods of improving the performance of the standard genetic algorithm. Three experiments will be performed, with each experiment testing if a particular change will improve the performance of the best GA found so far. In the first experiment we compare alternative methods of encoding the bit strings. We compare the commonly used binary encoding against gray scale encoding to determine if one is more preferable than the other (at least for our particular problem). In the second experiment we try an alternative method of selecting parents known as tournament selection and try to determine if it produces better results than the commonly used roulette wheel method. In the third experiment we try an alternative method for the progression of the GA. Generational progression in which one generation of specimen is completely replaced by the next is the most commonly used method. We compare this with a method referred to as steady state progression in which only one specimen in the population is replaced by a newly created one. Thus, there is no sudden turnover in the entire population, but rather a slow and gradual replacement. As each experiment is performed the best GA found as a result of that experiment will be used in the next experiment.

All three of these methods have been chosen based on current research going on in improving the performance of GAs. Also, all three of these methods have been used on other problems and have been found to improve the performance of the GA. Thus, we expect these changes to improve the performance of the GA for our particular problem of finding parameters for a RNN.

In the final experiment we use the best GA found to evolve a mixed size population and make a comparison of the results with the similar experiment performed using the standard GA.

4.2 Gray Scale Encoding

4.2.1 Introduction

One of the problems with binary encoding is that in some cases a large step must be taken in genospace in order to make a small move in phenospace. For example consider the bit string, 00111111; the 8-bit binary encoded representation of the decimal number 63. In order to change this to 64, the next higher point in phenospace, we must toggle the value of seven bits to produce 01000000. This is a significantly large move in genospace. It is highly unlikely that this change can be produced in one step by either the crossover or mutation operators.

Gray scale encoding has the property that in all cases to move to the next closest point in phenospace we only need to toggle one bit; the smallest possible move in genospace. This move could easily be made by the mutation operator. Thus, it is possible that this "smother" mapping between genospace and phenospace will allow the GA to converge to a solution faster and more often.

4.2.2 Experiment

The experiment to test gray scale encoding consisted of 50 trials of generating RNNs for each of the 4 network sizes (2, 3, 4, and 5 nodes in the second layer). The RNNs were again applied to the XOR problem as in Chapter 3. The setup for the GA was identical to that used in the third experiment described in Chapter 3 when fixed size networks were evolved, with the only difference being the use of gray scale encoding rather than binary. A population size of 30 specimen were used. The mutation rate was set to 0.01 and uniform crossover bias was 0.8 towards the parent with higher fitness. The results of this experiment can be compared directly with the results shown in Table 2 to determine which representation is better.

4.2.3 Results

The following table summerizes the results of the 50 trials for each of the 4 network sizes.

Table 4: Performance of the GA when gray scale encoding is used.

Number of nodes in second layer	Convergence ratio	Average number of generations
2	48:50	182.2
3	50:50	182.0
4	47:50	189.8
5	47:50	230.2

In comparing these results with those of Table 2, the most noticeable improvement produced by gray scale encoding seems to be a significantly higher convergence ratio for the smaller sized networks. It seems that the gray scale encoding does indeed cause the search space to appear to be smoother to the GA so that it is less likely for a parameter to get stuck due to a large move being required in genospace in order to make further progress. The effects of gray scale encoding appear to be more prominent in smaller sized networks because these networks have fewer parameters to adjust and if one gets stuck the GA may not be able to converge; whereas larger networks that have more parameters can adjust other parameters to compensate for the stuck parameter and thus can find a solution more often.

However, the number of generations needed to find a solution seems to have increased for the smaller sized networks. It should be noted that only the trials in which the GA converged to a solution are used in computing the average number of generations needed to find the solution. When binary encoding was used the trials which got stuck and did not find a solution were not included in the average. Only those trials which did not get stuck and were able to quickly find a solution were included. With gray scale encoding more trials are finding a solution because the parameters are not getting stuck and can eventually creep to the required value. Thus, trials which are taking longer, but never the less finding solutions are being included in the average and causing the average to be higher.

The results for both convergence ratio and speed seem to be about the same for the larger sized networks (4 and 5 nodes in second layer). This is due to the larger sized networks being less prone to getting stuck. The larger sized networks have more parameters that can be adjusted and if a particular parameter gets stuck other parameters can compensate for it.

Although gray scale encoding does not seem to indicate any substantial increase in how quickly the GA finds a solution, the significant increase in how often it is able to find a solution clearly indicates that gray scale encoding is preferable to binary encoding. In all subsequent experiments gray scale encoding will be used.

4.3 Tournament Selection

4.3.1 Introduction

Although the roulette wheel based parent selection method does guarantee that specimen with higher fitness are given greater opportunity to reproduce, it suffers from the fact that the opportunity it provides the specimen to reproduce is directly proportional to the fitness of that specimen relative to the average fitness of the population. Thus, in situations where the fitness of all the specimen is about the same with small deviations, the roulette wheel based selection method becomes almost equivalent to random selection. A rank based selection method such as tournament selection does not suffer from such situations. A rank based selection method also guarantees that specimen with higher fitness are given greater opportunity to reproduce. However, the amount of opportunity the specimen is given is determined only by the specimens rank and is independent of the specimens fitness relative to the average fitness of the population. It is expected that such a selection method will tend to be more greedy and can possibly find solutions more quickly at the expense of being more prone to getting stuck in local minima. The amount of greediness can be controlled by setting the tournament size. The larger the tournament size the more greedy the selection method.

4.3.2 Experiment

The experiment to test tournament selection consisted of 50 trials of generating RNNs for each of the 4 network sizes (2, 3, 4, and 5 nodes in the second layer). The RNNs were again applied to the XOR problem as in the previous experiment. The setup for the GA was identical to that used in the previous experiment. In fact the same algorithm (with gray scale encoding) was used with changes only to the selection method. The parent selection method was changed from roulette wheel to tournament selection with a tournament size of 2. That is, 2 specimen were selected randomly from the population. Each selection was independent of the other, so it was possible for both participants of the tournament to be the same specimen. The winner of the tournament was the specimen with the better fitness score. In case of a tie the specimen that was selected into the tournament earlier was chosen. The selection of the second parent was independent of the first. Thus, it is possible for both parents to be the same specimen. The results of this experiment can be compared directly with the results of the previous experiment shown in Table 4.

4.3.3 Results

The following table summerizes the results of the 50 trials for each of the 4 network sizes.

Table 5: Performance of the GA when tournament selection is used.

Number of nodes in second layer	Convergence ratio	Average number of generations
2	44:50	85.2
3	49:50	61.6
4	49:50	57.7
5	47:50	53.5

Comparing the results of this experiment with the previous experiment shows a substantial improvement in the average number of generations needed to find a solution. Using tournament selection has reduced the average number of generations in which a solution is found by more then 50% for all network sizes. This gain is achieved with only a small decrease in the convergence ratio for the smaller sized networks (2 and 3 nodes in second layer) and almost no significant loss for the larger sized networks (4 and 5 nodes in second layer).

A surprising result is that the average number of generations actually decreased as the network size got larger. In the previous experiments there was usually an increase with increasing network size. This result suggests that it's probably more easier for the larger sized networks to find a solution than originally thought.

It seems that at least for this problem the roulette wheel selection method is not making good use of the small differences in fitness between specimen and that the tournament selection method is much more preferable.

In all subsequent experiments the tournament selection method will be used.

4.4 Steady State progression

4.4.1 Introduction

Genetic Algorithms typically use generational progression in which a new generation of specimen are created from the current generation of specimen and the current population is completely replaced by the new population. However, it is not necessary that specimen from the current population be replaced only after an equal number of specimen to replace them are created. The replacement can be done whenever a certain number of specimen have been created. In steady state progression a specimen from the current population is replaced as soon as another specimen is created. The specimen replacement policy can be used to apply additional selective pressure. The higher level of iteration introduced by using steady state progression tend to make earlier use of better specimen and thus, allows a solution to be found more quickly. However, steady state progression will tend to be more greedy then generational progression and will improve performance only at the expense of being more prone of getting trapped in local minima.

4.4.2 Experiment

The experiment to test steady state progression consisted of 50 trials of generating RNNs for each of the 4 network sizes (2, 3, 4, and 5 nodes in the second layer). The RNNs were again applied to the XOR problem as in the previous experiments. The setup for the GA was also identical to that used in the previous experiments. The algorithm used in this test was the algorithm from the previous test (incorporating gray scale encoding and tournament selection) modified to use steady state progression instead of generational. Thus, one of the specimen in the population is replaced as soon as a new specimen is created. A tournament selection replacement policy is used. Two specimen from the current population are randomly selected to be participants in the tournament. The two selections are independent so that it is possible for both participants to be the same specimen. The winner is selected to be the specimen with the worse fitness score. In case of a tie the specimen that was selected into the tournament first is the winner. The winner is replaced by the newly created specimen, even if the winner has a better fitness value than the specimen replacing it. The results of this experiment can be compared directly with the results of the previous experiments to determine if steady state progression is beneficial.

4.4.3 Results

The following table summerizes the results of the 50 trials for each of the 4 network sizes.

Table 6: Performance of the GA when steady state progression is used.

Number of nodes in second layer	Convergence ratio	Average number of generations
2	43:50	67.2
3	50:50	41.8
4	49:50	36.1
5	47:50	42.7

With steady state progression there is no clear notion of generations, so we can only measure the total number of specimen evaluations that were performed in reaching a solution. However, by dividing the number of evaluations by the population size we can find an equivalent measure of generations that can be compared with previous experiments. Thus, the values in the column labeled average number of generations were produced by dividing the average number of evaluations required on trials that converged by the population size of 30 specimens.

Comparing the results of this experiment with the previous experiment shows a clear improvement in the average number of generations needed to find a solution for all network sizes. Using steady state progression has reduced the number of generations needed to find a solution by at least 20%. Surprisingly this gain in performance does not seem to impact the convergence ratio. Thus, solutions were found as often as in the previous experiment.

Unlike the previous experiment in which there was a continuous decrease in the average number of generations to find a solution with increasing network size, this experiment shows a decrease only until a network size of 4 nodes in the second layer. When 5 nodes are used in the second layer the number of generations required to find a solution increases. This could just be caused by a statistical sampling error. However, another possibility is that a maximum amount of selective pressure is being applied so that we are now seeing the expected minimum in the network size versus generations curve. This minimum is expected because for small network sizes, it may not even be possible to solve the problem with the number of free parameters, or there may only be a few possible solutions. Thus, resulting in a relatively larger number of generations needed to find solutions. For large networks just the greater number of free parameters that must be found makes the problem more difficult. Thus, one would expect there to be a network size that is best suited for solving the problem. The results of this experiment suggest that for the XOR problem the best network size is one that has 4 nodes in the second layer. It will be interesting to see if this holds true in the mixed size population experiment.

The significant gain in convergence speed without any decrease in the convergence ratio suggest that at least for this problem, steady state progression is superior to generational progression. Thus, in all subsequent experiments steady state progression will be used.

4.5 Mixed Size Population

4.5.1 Introduction

The fourth experiment discussed in Chapter 3 used a mixed size population to evolve not just the network parameters, but also the network architecture. The results of that experiment suggested that using a mixed size population improved the convergence ratio as well as the convergence speed. We would like to repeat this experiment using the enhanced version of the GA which incorporates gray scale encoding, tournament selection, and steady state progression. It will be interesting to see if using a mixed size population will improve the convergence ratio and convergence speed as suggested by the earlier mixed size population experiment. Also this experiment will serve as a test to confirm the results of the previous experiment which suggests that a network size with 4 nodes in the second layer should most often dominate the population.

4.5.2 Experiment

The mixed size population experiment consisted of 100 trials of generating RNNs for solving the XOR problem. A population size of 150 specimens was used with the initial population having an equal number (30 networks of each size) of networks with 1 to 5 nodes in the second layer. The setup for the GA was identical to that used in the fourth experiment of Chapter 3. However, the algorithm itself differed because it incorporated the three enhancements discussed in this Chapter. Also the parent selection method was slightly modified to allow for a mixed size population. The first parent was selected using a 2 player tournament. The second parent was also selected using a 2 player tournament, however, if the size (number of nodes in second layer) of the second parent was different than the first, another tournament was held to select the second parent. If after 10 tournaments a second parent is not selected, the first parent was also chosen to be the second parent. The results of this experiment can be directly compared with the results of the fourth experiment of Chapter 3 (table 3).

4.5.3 Results

The following table summerizes the results of the 100 trials and shows for each of the 5 network sizes, how often it dominated the population, how often it converged to a solution when it dominated, and the average number of generations it required to find a solution.

Table 7: Performance of the enhanced GA when a mixed size population is used

Number of nodes in second layer	Number of times dominated population	Number of times converged	Average number of generations
1	0	0	NA
2	15	15	18.1
3	21	21	16.2
4	37	37	13.1
5	27	27	24.9

Since the GA used steady state progression it was not possible to directly measure the number of generations needed in finding a solution. Instead the total number of specimen evaluations were measured and divided by the population size (in this case 150) to compute the equivalent number of generations. These values were used in computing the average number of generations for each of the network sizes.

These results show that using a mixed size population improved the convergence ratio for all network sizes. Whenever a particular network size dominated the population it was also able to converge to a solution. This maybe be due in part to a larger effective population size in the latter generations for the dominating network size. But the competition between network sizes also helps to weed out any network sizes that have gotten stuck in a local minimum. It is interesting to note that when the enhanced GA was used networks with 1 node in the second layer never dominated the population. This is probably due to the use of gray scale encoding in the enhanced GA which is allowing the other sized networks to find solutions more often.

The number of generations needed to find a solution is drastically reduced when using a mixed size population. This is due in part to interactions occurring only between smaller subpopulations of the larger population. Since the networks can only be crossed with others having the same size, the effective population size tends to be much smaller then the total population size of 150 specimens, particularly in the earlier generations. Thus, the total number of evaluations needed in finding a solution is only about twice as much as when a 30 specimen fixed size population is used. However, dividing this by the total population size of 150 causes the number of generations to be considerably less compared to the fixed size population.

The average number of generations for a network size of 5 is significantly larger then the other network sizes. This was found to be due to one trial which took over 45,000 evaluations compared to most other trials which took less then 3,000 evaluations. Since the termination criteria was still 150 generations or equivalently 75,000 evaluations this trial was not terminated earlier and treated as one that did not converge to a solution. Recomputing the average without including this trial gives a much more reasonable value of 14.1 as the average number of generations needed to find a solution for networks with 5 nodes in the second layer.

In comparing the results of this experiment with the results of the similar mixed size population experiment using the standard GA (table 3), it is clear that the combination of the enhancements made to the standard GA in this chapter have significantly improved its performance. Comparing table 7 with table 3 shows that the convergence rate has improved from about 120 generation to just approximately 18 generations; almost an order of magnitude less. Also the convergence ratio has improved from about 77% to 100%.

As expected from the results of the previous experiment, networks with 4 nodes in the second layer dominated most of the trials. Also the average number of generations needed to find a solution shows a minimum at a network size of 4. These results confirm that for the XOR problem networks with 4 nodes in the second layer are best suited to solve the problem.

4.6 Conclusion

Three experiments were performed in this Chapter to determine if certain modifications to the GA used in the previous Chapter would improve its performance. The experiments tested gray scale encoding, tournament selection and steady state progression. As each experiment was performed the best GA found as a result of that experiment was used in the next experiment.

From the results of these experiments it was found that gray scale encoding increases the convergence ratio (i.e. the number of times the GA converges to a solution), especially for the smaller sized networks. However, gray scale encoding does not seem to improve the speed of convergence (i.e. the total number of generations required by the GA to find a solution). Using tournament selection as opposed to roulette wheel selection significantly increases the performance of the GA by reducing the number of generations needed to reach a solution. This increase in performance is gained with only a small decrease in the convergence ratio. Using steady state progression also helps to improve the speed of convergence, but Surprisingly does not seem to effect the convergence ratio. The best GA found as a result of these three experiments was significantly better than the original GA used in Chapter 3.

A final experiment was performed using a version of this enhanced GA, modified to allow mixed size populations, to test if further improvements could be gained by using a mixed size population rather than a fixed size population. The results showed improvements in both the speed of convergence and the convergence ratio. This experiment also showed that networks with 4 nodes in the second layer are best suited for solving the XOR problem.

The amount of improvement between the original GA and the enhanced GA being almost an order of magnitude better suggests that at least the combination of these enhancements have produced a GA that is in general better than the original. Thus, the enhanced GA found as a result of these experiments will be used in all subsequent experiments.

Chapter 5
Learning Finite State Machines

5.1 Introduction

Thus, far we have only applied RNNs to the XOR problem. Although this problem is nontrivial, it can be solved by networks that use only a feedforward architecture and does not require a RNN. This is because the output of the network depends only on the current inputs being applied. If a problem required the outputs to depend not only on the current input, but also on past inputs in such a way that the network had to maintain an internal state, it could not be solved by a network using a simple feedforward architecture. A finite state machine requires such a behavior and thus, would require a RNN to learn not only its input-output mapping, but also the internal states it uses.

In this chapter we apply the best genetic algorithm found in the previous chapter to the problem of learning a finite state machine (FSM) from examples. In this problem the neural network is presented with a sequence of input patterns and the output of the network is checked after each pattern is applied to determine how close it is to the desired output and thus, determine the fitness of the network. The same input pattern, however may require a different output depending on what the previous inputs have been.

Two particular FSMs will be used to evolve RNNs that can mimic them. The first FSM is one that is able to distinguish if a binary sting belongs to the Tomita #4 grammar or not. The Tomita #4 FSM starts out producing an output of 1 and maintains that output until a sequence of three consecutive 0's are seen on the input. When the third 0 appears the output of the FSM changes to a 0 and stays that way forever regardless of future input. The RNN needed to mimic this FSM would need only one input and output and at least one or more hidden node. The second FSM is one that determines the parity of a bit string. The parity FSM starts with an output of 0 and counts the number of 1's that have appeared on the input and produces an output of 0 if this number is even and 1 if it is odd. The RNN required for this would also require only one input, one output and at least one hidden node. Both problems require the resulting RNN to be able to maintain an internal state and produce an output that is dependent on both this state and the current input.

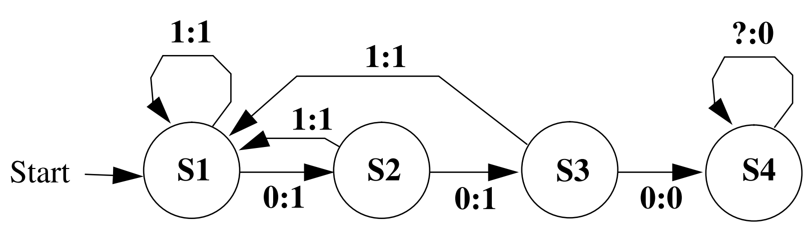


Figure 5: A state diagram of the Tomita #4 FSM. The number before the colon is the input which causes that link to be followed and the number after the colon is the output the state machine produces.

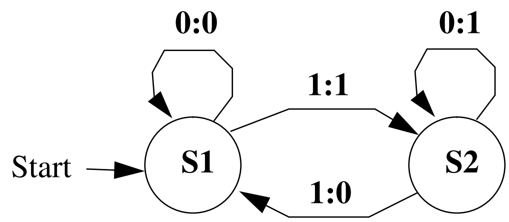


Figure 6: A state diagram of the Parity FSM. The number before the colon is the input which causes that link to be followed and the number after the colon is the output the state machine produces

5.2 Network Structure and Fitness

Since both finite state machine problems used in this Chapter are binary in nature we have chosen to use a discrete time RNN. The network architecture, the activation function and the range of the parameters are the same as described in Chapter 1.

To determine the fitness of a particular network at simulating the given FSM, 40 input sequences of length 10 are used to test the network. The 40 sequences are picked randomly, but the same 40 sequences are used to test each network. Before presenting each sequence the network is initialized so that the internal state of each node in the second layer is 0.0 and the output of the nodes is computed from this state. As each input pattern in the sequence is applied to the network, the network is simulated for 2 iterations before its output is checked. The fitness of the network is measured after each pattern in the input sequence is applied and the fitness values are averaged together to find the networks fitness for the given sequence. The procedure is repeated for each of the 40 sequences to compute the networks fitness on each sequence and the resulting values are averaged together to get the final fitness of the network. The following equation was used to compute the fitness:

$$fitness = \frac{1}{SL}\sum_{i}^{S}\sum_{j}^{L} 1 - (d_{ij} - o_{ij})^2$$

d_{ij} is the desired output on pattern j of sequence i and o_{ij} is the actual output. S is the number of sequences and L is the length or number of patterns in each sequence.

Unlike the XOR problem in which all combinations of the input patterns could be presented to the network, all combinations of input sequences cannot in general be presented for the FSM problems. Even when the length of the sequence is limited to just 10 patterns with each pattern being just one bit, there are 1024 sequences possible. Thus, the 40 randomly chosen sequences represent only a small fraction of the input space.

It is possible that the networks found using this inexhaustive fitness function have learned to produce the appropriate output at the appropriate time to mimic just the 40 sequences they were trained with and have not really learned to simulate the behavior of the corresponding FSM. Thus, the resulting networks must be tested on a set of novel sequences to determine what they have really learned.

5.3 Experiments and Results for the Tomita #4 FSM

5.3.1 Initial Results

In the first experiment the DTRNN and fitness function described in the previous section were applied to the Tomita #4 FSM problem. 100 trials were performed using a mixed size population in which the number of nodes in the second layer could range from 1 to 5. The starting population in each trial had 30 specimen from each of the possible network sizes. Runs were terminated if a specimen with a fitness greater than 0.99 was not found in 200 generations. The following table shows the results of this experiment.

Table 8: Results of initial Tomita #4 FSM experiment.

Number of nodes in second layer	Number of times dominated population	Number of times converged	Average number of generations
1	23	0	NA
2	19	2	126.1
3	24	10	74.8
4	16	14	57.8
5	18	18	46.6

The smaller sized networks (1-3 nodes in second layer) dominated the population most often in these trials, but were not even able to find solutions 25% of the time. The larger sized networks (4-5 nodes in second layer) found solutions almost every time they dominated the population. Also the larger sized networks took significantly fewer generations to find solutions.

A few of the evolved networks were tested with novel input sequences to determine if they had really learned the FSM or just the 40 sequences on which they were trained. The networks were tested on 1400 sequences of length 100 and on 140 sequences of length 1000. Every network tested produced a fitness greater then 0.99 on both tests. Even though the networks were trained on 40 sequences of length 10, they were able to learn the underlying FSM. Surprisingly even some networks with only 2 nodes in the second layer were able to learn the Tomita #4 FSM. When the actual output of the networks were observed it was found that sometimes the network produced values like 0.7 or 0.3 when it should have been producing an output of 1 or 0 respectively. This occurred more frequently for networks with fewer nodes in the second layer. However, in all cases just rounding the number to the closest integer resulted in the desired output.

The fact that the networks are able to produce the correct output on sequences that are much longer then those on which they were trained is an indication that the networks are forming very stable internal states. Using an activation function that has hard limiting saturation points seems to be the critical element in allowing networks to form such stable internal states. If a sigmoid activation function had been used the internal states formed by the networks would not have been stable and the network would not have produced the correct output on sequences that were much longer than the training sequences [Zeng, Goodman and Smyth, 1993]. This is one example in which a linear hard limiting activation function is preferable to the commonly used sigmoid activation function. The GA is well suited to handle such activation functions since it does not make use of any gradient information and thus, does not require the activation function to be continuous or differentiable.

5.3.2 Complementary Clock Inputs

When FSMs are implemented in hardware, some form of a clock signal is used to drive the FSM. The clock signal synchronizes the components in the FSM so that the clock signal is what triggers the FSM to move from one state to the next. In the second experiment we provided 2 additional inputs to the network to determine if this would improve the convergence rate or convergence ratio. One of the inputs started out high (1) for half the clock cycle and went low (0) for the remaining half of the cycle. The other input was Complementary to the first, so that it started out low and went high after half the cycle. The cycle length was 2 simulation steps. Thus, the network was simulated for the same number of steps between presentation of input patterns as in the previous experiment. The following table shows the results of this experiment.

Table 9: Results of using Complementary clock inputs on the Tomita #4 FSM.

Number of nodes in second layer	Number of times dominated population	Number of times converged	Average number of generations
1	19	0	NA
2	21	4	72.3
3	25	18	46.0
4	19	18	39.9
5	16	16	31.5

Adding the Complementary clock signals significantly improves the convergence rate for all network sizes. The convergence ratio for the smaller sized networks is also greatly improved. The networks were not analyzed to see how the clock signals were being used, since this was not the goal of our experiment.

Some of the evolved networks were tested with novel input sequences to determine if they had really learned the FSM. The networks were tested on 1400 sequences of length 100 and on 140 sequences of length 1000. Every network tested produced a fitness greater then 0.99 on both tests.

5.3.3 Sequential Biased Fitness Function

In the third experiment we make use of the sequential nature of the input sequences to try and improve the fitness function in hopes of further improving the convergence rate and ratio. It is hypothesized that the output of the network for different patterns in the sequence should not be considered equally. More weight should be given to the output produced for the earlier part of the sequence than for the later part of the sequence. Due to the sequential nature of the problem once a mistake has been made on one pattern, producing the correct output on latter patterns is irrelevant. Thus, a network which can produce the correct output for only the first 3 of the 10 patterns in a sequence should be considered better than a network which can produce the correct output for 8 of the 10 patterns, but makes a mistake after the first 2 patterns.

The fitness function was changed to the following to incorporate this sequential bias in the final fitness.

$$fitness = \frac{1}{SL} \sum_{i}^{S} \sum_{j}^{L_i} 1 - (d_{ij} - o_{ij})^2$$

L_i is the location in sequence i where the first mistake is made by the network. Thus, the network is scored as if it had made a mistake on all patterns after L_i. The first mistake is considered to have occurred when $|d_{ij} - o_{ij}| > 0.5$.

The two Complementary clock inputs which proved to be advantageous in the previous experiment will also be used in this experiment. Thus, this experiment will differ from the previous only in the fitness function used. The results of this experiment can be directly compared to the results of the previous experiment.

The following table shows the results of this experiment. Some of the resulting specimen were tested as in the previous experiments and confirmed that they had indeed learned the FSM.

Table 10: Results of using a sequentially biased fitness function

Number of nodes in second layer	Number of times dominated population	Number of times converged	Average number of generations
1	11	0	NA
2	25	11	47.1
3	32	26	40.3
4	17	17	42.0
5	15	15	57.4

Using a sequentially biased fitness function has significantly improved the convergence ratio for the smaller sized networks. The larger sized networks already had an almost perfect convergence ratio since the first experiment. But now with the modified fitness function the larger sized networks have a 100% convergence ratio. The overall convergence ratio has increased to almost 70% from a value of about 56% in the previous experiment.

The results for the convergence rate are somewhat mixed. There is a definite improvement in convergence rate for the smaller sized networks. However, there seems to be a decrease in the convergence rate for the larger sized networks. The reason for these results, it is hypothesized, stems from the fact that although the original fitness function and the sequentially biased fitness function have the same global optimums in fitness space, the shape of the fitness landscape imposed by each is different. The sequentially biased fitness function provides a more directed landscape leading towards the global optimum. Although this helps guide the potential solutions along a path that presumably contains less local optimums, it is less efficient than being able to directly approach the global optimum. Because the smaller sized networks are more easily trapped in local optimums they benefit from a directed fitness space as evident by the improved convergence ratio and convergence rate. However, the larger sized networks which were not being trapped by the local optimums and were able to take a more direct route to the global optimum are now slowed down by the longer route they must take in the directed fitness space as evident by the decrease in convergence rate.

5.4 Experiments and Results for the Parity FSM

The desired output of the Tomita #4 FSM does not tend to be very random. Rather it is composed of a series of 1s followed by a series of 0s. Thus, it is possible that the improvements suggested by the experiments in the previous section were specific to the nature of the Tomita #4 FSM and may produce different results for other FSMs. In order to verify the results of those experiments a second problem was chosen. The Parity FSM problem was selected since the desired output of this FSM tends to toggle much more and can start with either a 0 or 1. The same three experiments conducted in the previous section were repeated for the Parity FSM problem. The following tables show the results for each of these experiments.

Table 11: Results of initial experiment

Number of nodes in second layer	Number of times dominated population	Number of times converged	Average number of generations
1	43	0	NA
2	17	0	NA
3	19	9	43.8
4	13	9	41.5
5	8	8	38.7

Table 12: Results of using complementary clock inputs

Number of nodes in second layer	Number of times dominated population	Number of times converged	Average number of generations
1	28	0	NA
2	28	8	62.1
3	12	11	24.6
4	20	18	48.4
5	12	11	40.6

Table 13: Results of using a sequentially biased fitness function

Number of nodes in second layer	Number of times dominated population	Number of times converged	Average number of generations
1	0	0	NA
2	19	13	31.7
3	32	25	25.7
4	29	29	26.7
5	20	20	31.7

The initial experiment shows that there is a very strong tendency for networks with only one node in the second layer to dominate the population. Although these networks are unable to solve the problem, they are able to produce a higher fitness in the early generations and dominate the population. The fewer number of generations needed to find solutions for this problem as compared to the Tomita #4 FSM problem suggests that this is an overall easier problem. This can be expected since the minimum number of states required to implement the Parity FSM is only 2 as compared to 4 for the Tomita #4 FSM.

As in the Tomita #4 experiment adding Complementary clocks inputs dramatically improves the overall convergence ratio. However, it is not clear weather it helps improve the convergence rate for this problem. Although there was significant improvement for networks with 3 nodes in the second layer, there was a slight decrease for the larger sized networks.

Using a sequentially biased scoring function significantly improves both the convergence rate and ratio. Also it completely eliminated networks with only one node in the second layer from dominating the population. From our experience with the Tomita #4 problem an improvement in the convergence ratio was expected, however an improvement in the convergence rates was not expected. However, a possible reason for the improvement could be that the fitness landscape for the Parity FSM problem contains more local optimums and slows down direct progress towards the global optimum. Although the directed path may be longer the rate at which potential solutions move along it may be faster, resulting in an improved convergence rate for all network sizes.

Some of the evolved networks were tested with novel input sequences to determine if they had really learned the parity FSM. The networks were tested on 1400 sequences of length 100 and on 140 sequences of length 1000. Every network tested produced a fitness greater then 0.99 on both tests.

The results for the Parity FSM problem are quite similar to the results obtained for the Tomita #4 FSM problem. The combination of adding complementary clock inputs and using a sequentially biased fitness function produced significant improvements in both the convergence rate and ratio. The fact that the results were similar for two very different FSMs suggests that these modifications do produce improvements in general for FSM problems.

5.5 Discussion of Results

The experiments described in this chapter have demonstrated that RNNs are capable of learning FSMs from relatively short and few examples of input-output pattern sequences. Both the Tomita #4 and the Parity FSM learning problems can even be solved by networks with only two nodes in the second layer. Some of the resulting networks were tested and found to be stable on many input sequences that were orders of magnitude longer that those used when evolving the networks. The reason for the networks being able to form stable states is due to the piecewise linear activation function used by the nodes in the second layer. Such activation functions can be quite easily handled without any modifications to the GA.

A practical lesson learned from our experience with the FSM problems is that incorporating some apriori knowledge about the problem into the fitness function or in the inputs to the network can produce a significant improvement in how quickly and how often solutions are found. For both problem the initial application of the GA produced only marginal results. However, using the same GA, but with additional complementary clock inputs to the networks and a sequentially biased fitness function produced much better results.

Chapter 6
Balancing an Inverted Pendulum

6.1 Introduction

In the previous chapter we demonstrated that RNN are capable of learning FSMs from example. In this chapter we apply the GA to the task of developing a recurrent neural network controller capable of balancing an inverted pendulum. This task requires continuous valued inputs and outputs, unlike the discrete valued problems of the previous chapter. Thus, the nodes in the second layer will use the continuous time formulation for their dynamics. The task of balancing an inverted pendulum can be solved by a feedforward network and does not require a RNN. However, we anticipate from our experience with the XOR problem that the solutions found will make use of the recurrent connections and will be able to solve the problem using fewer hidden nodes than would otherwise be possible.

We will again be using the best GA found in Chapter 4. The use of a GA will allow us to specify a fitness function which does not directly incorporate the required output of the network for the current inputs. We would like to see if suitable networks can be evolved with a fitness function that uses only indirect measurements of fitness such as how long the pendulum was balanced or the position of the pendulum and does not use direct information like what force to apply to the cart. Thus, the networks are evolved by specifying what they must do and not how they must do it.

The cart and pendulum system was represented as shown in the following figure.

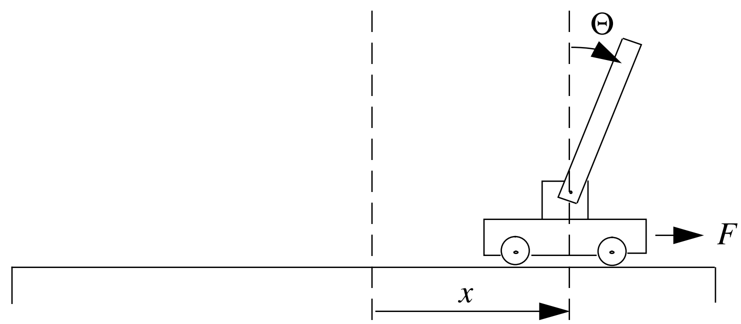


Figure 7: The cart and inverted pendulum system. The network controlling the cart receives the position and velocity of both the cart and pendulum and must produce the force to be applied to the cart as output.

The cart and pendulum system is modelled by the following equations:

$$\ddot{\Theta} = \frac{g \sin\Theta + \cos\Theta \left[\frac{-F - ml\dot{\Theta}^2 \sin\Theta + \mu_c \operatorname{sgn}\dot{x}}{m_c + m} \right] - \frac{\mu_p \dot{\Theta}}{ml}}{l \left[\frac{4}{3} - \frac{m\cos^2\Theta}{m_c + m} \right]}$$

$$\ddot{x} = \frac{F + ml\left[\dot{\Theta}^2 \sin\Theta - \ddot{\Theta}\cos\Theta \right] - \mu_c \operatorname{sgn}\dot{x}}{m_c + m}$$

The values used for the constants are: g = -9.8 m/s^2, acceleration due to gravity; m_c = 0.5 kg, mass of cart; m = 0.1 kg, mass of pole; l = 0.5 m, half-pole length; μ_c = 0.0, coefficient of friction of cart on table; μ_p = 0.0, coefficient of friction of pole on cart. The force applied to the cart was limited in the range of [-10, 10] Newtons.

6.2 Network Structure and Fitness

The network architecture, the activation function and the range of the parameters are the same as described in Chapter 1. continuous time nodes were used in the second layer. The networks received four inputs and produced one output. The inputs to the network were the position and velocity of both the cart and pendulum. The output of the network was the force to be exerted on the cart. The saturation range for the inputs was [-1.5, 1.5] radians, [-6, 6] radians/second, [-10, 10] meters, and [-20, 20] meters/second. The [0, 1] range of the output node was mapped to [-10, 10] Newtons.

Various attempts were made at finding a fitness function which could successfully evolve stable RNN based controllers. Initially time was used as the measure of fitness. The score of a network was simply the number of simulation steps it could "balance" the pendulum. By balance we mean keeping the angle of the pendulum and the position of the cart within specified limits. The limits used were ± 90 degrees for the angle and ± 10 meters for the position. It was found that this measure of fitness was not sufficient to produce any suitable networks. Other fitness functions based on just the angle of the pendulum or the position of the cart and various combinations of the two were also tried and found to be unsuccessful. Using a combination of both time and position of the pendulum, finally turned out to be a good measure of fitness capable of producing stable controllers.

The function used to determine the fitness of a particular network at balancing the inverted pendulum was defined as:

$$fitness_s = N_b - \frac{1}{2}\sum_{i=1}^{N_c}\left[\frac{x_i}{x_{max}} + \frac{y_i}{y_{max}}\right]$$

$fitness_s$ is the fitness of the network on a particular simulation. N_b is the number of simulation steps the pendulum was balanced before falling out of limits. x_i and y_i are the x and y distances (positive) of the tip of the pendulum from the desired position. x_{max} and y_{max} are the maximum values possible for x_i and y_i respectively (which in this case are 10 and 1). Thus, on each simulation step the network gets a value between 0 and 1 based on the distance of the tip of the pendulum from the desired position. However, this equation only applies as long as the network is able to balance the pendulum. The maximum fitness a network could receive on a particular simulation is the total number of steps in the simulation. The overall fitness of the network is defined to be the sum of its performance on each of the simulations.

$$fitness = \sum_{i=1}^{S} fitness_s$$

In the experiments described later 10 simulations were used to evaluate each specimen. The cart was always started from the center of the table, but the angle of the pendulum was started at values of: 10, -10, 15, -15, 20, -20, 25, -25, 30 and -30 degrees. The simulations were run for maximum corresponding times of: 1, 1, 3, 3, 5, 5, 7, 7, 9 and 9 seconds. Thus, the length of the simulations were longer if the initial position of the pendulum was further from center. Using these maximum time limits the total time a network would be able to balance the pole is 50 seconds. Since a delta time of 0.01 seconds is used for the simulation steps, this means the maximum fitness a network could achieve is 5000. However, it is important to note that even the best network will not achieve this maximum fitness score. Even if a network was able to balance the pole extremely well and keep it at the desired position, the network will still not get a perfect score of 1 on some of the initial simulation steps when it is brining the pole from its initial position to the desired position.

The networks were initialized at the beginning of each simulation so that the internal state of each node in the second layer was set to 0 and the output set to the corresponding value determined by the activation function.

6.3 Experiments and Results

One hundred trials were run using a mixed size population with 1 to 5 nodes in the second layer. A total of 150 specimen were used in the population with 30 specimen of each network type. Since it is difficult to decide when a network has learned to balance the pole based on the fitness of the network, all runs were evolved for 200 generations or equivalently 30,000 specimen evaluations. Only the score at the end of the run was saved for the results. The results of these experiments are summarized in the following table.

Table 14: Results for the inverted pendulum problem.

Number of nodes in second layer	Number of times dominated population	Minimum fitness	Maximum fitness	Average fitness	Standard deviation in fitness
1	20	1257.7	4294.8	1977.4	861.4
2	24	1475.8	4724.4	2985.7	1210.1
3	22	1656.8	4814.2	4057.2	979.9
4	20	1426.0	4859.0	4433.7	890.6
5	14	4591.6	4864.8	4778.5	69.2

The networks evolved were tested to determine if they were stable controllers for pole angle perturbations in the range [-30, 30] degrees. It was found that networks with a fitness greater than 4700 were definitely stable controllers. Networks with a fitness less than about 4250 were definitely not stable. The networks with a fitness between these limits displayed oscillatory behavior. The behavior of some of these networks is shown below. The solid line in each figure shows the x coordinate of the tip of the pendulum and the dotted line shows the x coordinate of the cart.

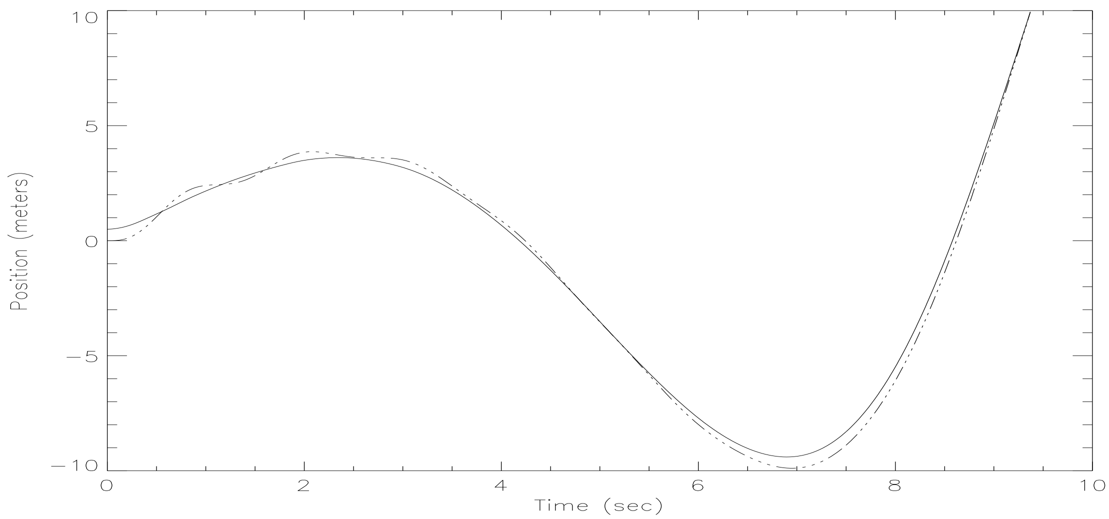


Figure 8: An unstable controller with a fitness of 3910 and 2 nodes in the second layer. The solid line shows the position of the tip of the pendulum and the dotted line show the position of the cart.

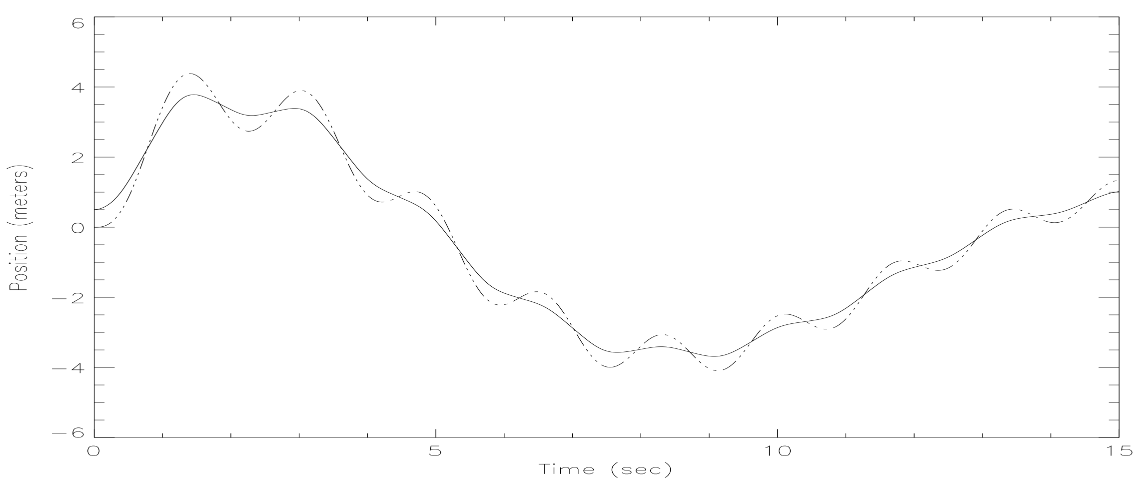


Figure 9:A stable, but very oscillatory controller with a fitness of 4315 and 3 nodes in the second layer.

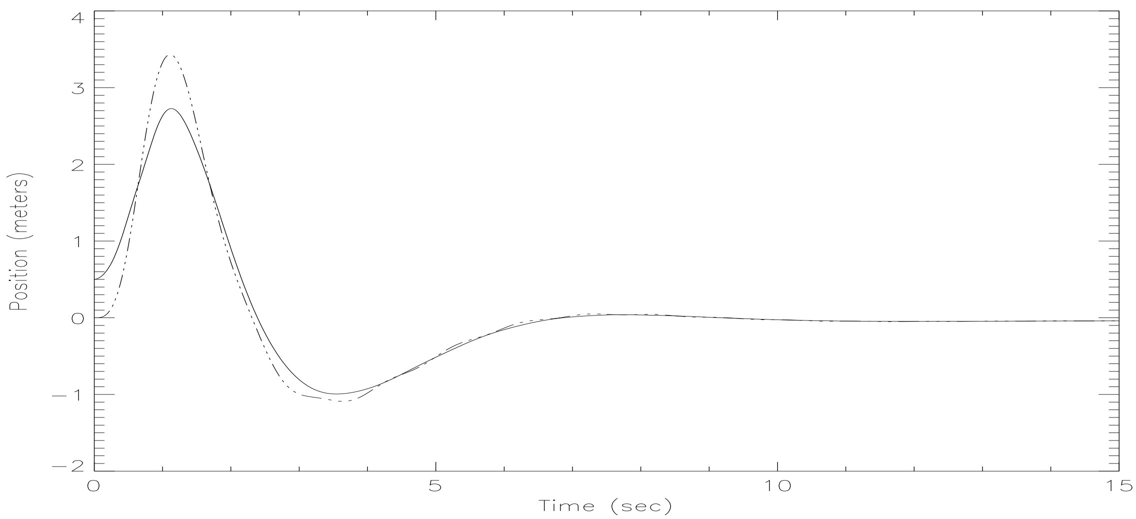


Figure 10:A stable controller with a fitness of 4814 and 3 nodes in the second layer.

6.4 Discussion of Results

One of the most surprising results of these experiments was the discovery of stable networks with just one node in the second layer. This network would be equivalent to a two layer feedforward network with four nodes in the input layer and one node with self feedback in the output layer. Although these networks were stable, their behavior was very oscillatory. Only two such networks were found in the 20 times the population was dominated by networks with one node in the second layer. Both networks had similar sets of weights and thresholds. A graph of the behavior of one of these networks is shown below.

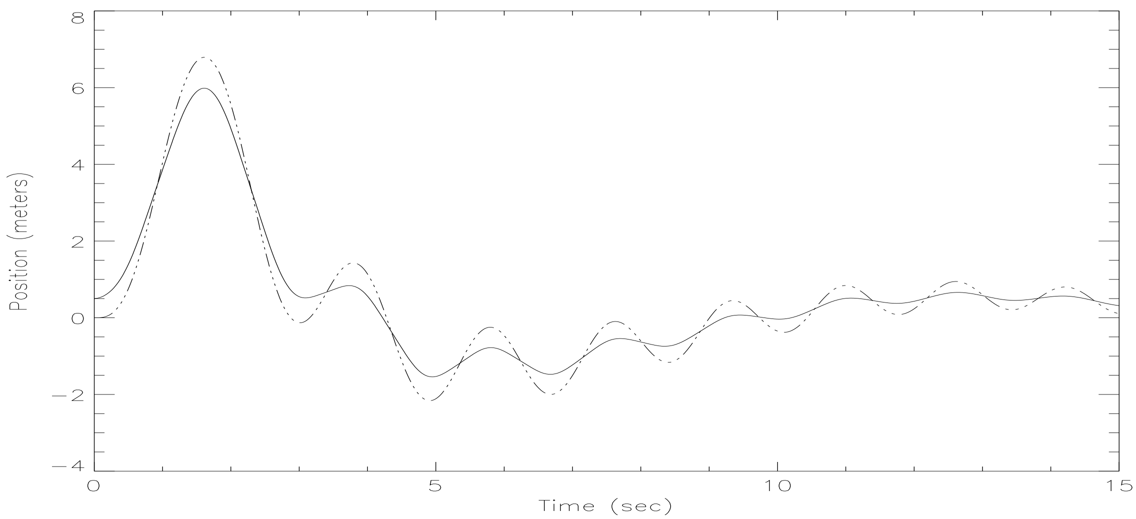


Figure 11: A stable but oscillatory controller with a fitness of 4295 and just 1 node in the second layer.

As can be seen from the results of this experiment, the inverted pendulum problem is significantly more difficult than the previous problems. Stable networks were found only 54% of the time. However, this may not completely be the fault of the GA. It may be that the inverted pendulum problem is very difficult when limited to networks with just 1 to 5 nodes in the second layer. If the initial population had been seeded with larger networks we may have found stable networks more often. The fact that stable networks were always found whenever the population was dominated by networks with 5 nodes in the second layer supports this hypothesis. However, the GA we used is still at some fault for not having allowed the networks with 5 nodes in the second layer to dominate more often. The problem is caused by the smaller sized networks being able to improve their fitness more quickly in the early generations and dominating the population before the larger sized networks are able to catch up. A possible way to fix this problem may be to allow the different sized networks to evolve independently during the early generations.

The use of a fitness function which incorporated both the amount of time the pendulum was balanced and the position of the pendulum in the evaluation of the fitness was critical to the success of finding solutions to this problem. Attempts at using fitness functions which used only one or the other of these measures did not produce any stable (or even oscillatory, but stable) networks. Although it may be that all of these fitness functions have the same global optimum, it seems that using a combination of both time and position in the evaluation makes the search space much easier to traverse than when only one or the other is used.

Chapter 7

Discussion and Further Work

7.1 Introduction

The intent of this thesis was to investigate the GA as a means of finding network parameters and architecture for a RNN and also to gain experience in applying GAs to RNNs. We will first discuss the results of the experiments performed in this thesis in relation to these goals. Next we will discuss some lessons learned in applying GAs to RNNs. Finally we will discuss some future work that can be done to possibly further enhance the GA as a means for evolving RNNs.

7.2 Applicability of GAs to RNNs

There are two areas in which the applicability of the GA for evolving RNNs were explored. The first was for finding the network parameters of the RNN. The other was for finding the right network architecture to use for the given problem.

The standard GA, with its main characteristics being generational progression and roulette wheel selection, was found to be suitable for finding the network parameters of RNNs applied to simple problems such as the XOR problem. However, it was found that the performance of the standard

GA could be improved significantly by making a few changes to the algorithm. Using steady state progression, tournament selection, and gray scale encoding all helped improve the performance of the standard GA. The combination of these three improvements decreased the convergence rate from about 170 generations to about 50 generations and increased the convergence ratio from about 70% to greater than 95% on the XOR problem. This enhanced GA was applied to the more dynamic and complex problems of finding a RNN for a given Finite State Machine (FSM) and for finding a network to balance an inverted pendulum. The GA proved to be quite successful at solving both of these problems. However, it was found that a well defined fitness function was very critical to the success of the GA on these problems.

As discussed earlier the GA is inherently not well suited for evolving the network architecture. This is due primarily to the fixed length bit string representation required by GAs. The approach taken in this thesis to alleviate this problem was to seed the initial population with specimens of various network architectures and to modify the parent selection scheme so that only compatible parents were selected for crossover. The network architecture of specimens was varied by changing the number of nodes in the second layer. It was found, as expected, that when such mixed architecture populations were evolved, one of the network types would slowly dominate the population. The network architecture which was better suited for solving the problem was most likely to dominated the population. As discussed in Chapter 3, the total amount of computation required to find a suitable network architecture using this method is less than would be required by running several simultaneous GAs each using a population with a different network architecture.

In conclusion the GA has proven to be a very powerful and promising method for finding both the network parameters and the network architecture. Some of its benefits of using a GA include the ability to easily handle different neuron models incorporating non conventional activation functions or other parameters in addition to the usual weights and thresholds. Another appealing feature of the GA is the inherently parallel nature of the algorithm. The evaluation of specimens, which is the most computationally intense part of the algorithm, can easily be spread over multiple processors with each processor evaluating a given specimen. This may allow for a linear speedup with the number of processors.

7.3 Practical Issues of Applying GAs

The GA is not a very rigorously defined algorithm with respect to the details of its implementation. This has both advantages as well as disadvantages. The advantage is that it allows the algorithm to be flexible so that its implementation can be modified to fit a wide range of problems. The disadvantage is the details of its implementation may need to be modified to get good performance on a specific problem. Our experience with the GA has shown that even a small change in the algorithm, such as the mutation rate, selection scheme, etc., can lead to a significant difference in its performance. As mentioned earlier, the three modifications which were made to the standard GA, in this thesis, lead to significant improvements in performance. However, there may still be other modifications which could be made to further improve the algorithms performance.

Our experience from the experiments performed in this thesis has shown that the performance of the GA is also very much dependent on the fitness function chosen to define the problem. Although two fitness functions may have the same global minimum in theory, one may be much easier for the GA to traverse than the other. In the finite state machine problems it was found that making the fitness function sequentially biased (to reflect the sequential nature of finite state machines) helped improved performance. Also adding additional clock inputs, which provided no additional information about the problem to be solved, helped to improve the performance. On the inverted pendulum problem, fitness functions based on just the length of time the pendulum was balanced or just the position of the cart and pendulum were found not to be suitable. It was found that the fitness function needed to combine both the length of time the pendulum was balanced and the position of the cart and pendulum in order to find solutions on even some of the trials.

In conclusion our experience with the GA has shown that it is not trivial to apply the GA to a particular problem. The procedure for applying the GA is not a cook book recipe. It requires a significant amount of effort in first tweaking the various methods and parameters of the GA to get satisfying results. In addition the fitness function defining the problem may need to be experimented with to further improve performance. However, these efforts will pay off if other easier methods for solving the problem are not available or if the final intent is to solve the problem using a parallel approach. Also the implementation of a GA is generally easier than that of other complex optimization algorithms. Thus, it requires less effort to get started.

7.4 Future Work

The flexible nature of the GA is well suited for experimenting with new ideas for possibly improving the performance of the algorithm. We divide the discussion of possible future work into three areas. First we discuss other possible enhancements that could have been tried in improving the performance of the standard GA. Next we discuss the possibility of including Larmarkism into the GA so that improvements could also be made during the evaluation of the specimen and incorporated back into the genome. Finally, we discuss the possible use of variable length genomes and alternative genome representations to provide a more natural evolution of the network architecture.

7.4.1 Finding Better GA Parameters and Methods

There are many more possible enhancements that could have been tried in this thesis, but were not, due to limited time constraints. Some of these possible enhancements include:

Experimenting with different population sizes. For the experiments done in this thesis a population size of 150 specimen was used for mixed network architecture runs and a population size of 30 was used for fixed network architecture runs. Other population sizes could be tried in an attempt to find a population size that allows for better overall performance. However, the best population size may be very sensitive to the specific problem. Thus, a population size which produces good results for the XOR problem may not be the best size for the inverted pendulum problem.

Experimenting with the replacement policy in steady state progression. Although we chose a two player tournament selection replacement policy for deciding which member of the current population would be replaced by a newly created one, other policies such as random selection, roulette wheel selection, or other tournament based selections could be tried.

Experimenting with other crossover operators. The parameterized uniform crossover operator was used in all experiments based on its merits discussed in the GA literature. However, for some problems it may be that one of the other crossover operators such as single point, or multi-point crossover may be better suited.

When mixed architecture networks are used, the size of the overall population could be reduced as a particular network size dominates the population. This could possibly allow for the algorithm to find a solution with even less computational effort. However, reducing the size too quickly may cause the GA to get stuck at a local minimum. Thus, finding a good reducing schedule would be critical to this possible enhancement.

Also when mixed size populations are used the different network sizes could be kept isolated during the first N generations as if multiple simultaneous independent GAs were being run. The different network sizes could then be allowed to compete for dominance of the population after the Nth generation. This would help prevent a network size that is not well suited to solve the problem from prematurely dominating the population and would thus help improve the convergence ratio.

7.4.2 Incorporating Lamarckism

One interesting possible enhancement that should be tried in the future is that of including Lamarckism in GAs. Lamarckism generally refers to phenotypic changes in a specimen which occur during its lifetime being passed back to its genome and eventually to its offsprings. In biological organisms, it is believed that Lamarckism does not occur, however there is no reason why it could not occur in artificial organisms. In our case of evolving

RNNs this would mean that we allow for an on line training algorithm, such as the Alopex algorithm [Venugopal, Pandya, and Sudhakar, 1994], to operate on the RNN during its fitness evaluation and make small changes to the network parameters. At the end of the fitness evaluation we must create a new bit string genotype of the specimen reflecting its modified network parameters. The new bit string would be used if this specimen was selected for reproduction. The idea of using Lamarckism with GAs was first suggested by [Hinton and Nowlan, 1987]. Their experimentation with a very simple problem showed that Lamarckism could assist the GA in finding the global minimum.

7.4.3 Variable Length Genome Representations

The approach introduced in this thesis for selecting a network architecture is based on the fixed length genome requirement imposed by the standard GA. However, if in the future we want to truly allow for an evolutionary process to operate on the architecture of the network as well, we must allow for non-fixed length genomes and for operators which allow for these genomes to be combined to produce new genomes which may have lengths very different from the parents. This would allow for novel network architectures to emerge aside from the ones with which the population was initially seeded. If we decide to accept these changes to the GA, then the remaining difficulty is the encoding mechanism to use for encoding the network parameters on the genomes. In the standard GA the position of the parameter on the genome determines it function. However, in a variable length genome representation the location of the parameters will change if a genome gets bigger or smaller. Thus, each parameter must be able to represent its function independent of the position at which it occurs. Also, since the crossover operators will generally not know about the details of the encoding mechanism used, it is possible for their results to produce over specified or under specified (with respect to the network parameters needed) offsprings. An approach referred to as messy GAs tries to address these problems caused by variable length genomes [Goldberg, Deb, and Korb, 1991]. However, an alternative possibility, particularly well suited for the evolution of neural networks is to let the genome represent not the network parameters, but rather instructions to a cellular automaton. Thus, from a single cell executing the instructions provided by the bit string, a network could emerge after many iterations of cell division and growing connections. Although this approach is much more closer to what happens in nature during the genotype to phenotype mapping, it is computationally much more expensive. Also, Lamarckism could not be incorporated into the GA with this method, since an inverse mapping from the phenotype to the genotype does not seem possible with this representation.

7.5 Conclusion

RNNs hold the promise of providing a more general network architecture that is applicable to a wider range of dynamic and state dependent problems. However, finding the network parameters as well as the right network architecture for the problem is generally more difficult when a RNN is used. The results of this thesis suggest that an evolutionary approach such as a GA can serve as a very powerful tool in helping to find the network parameters and architecture for applying a RNN to a specific problem. Ideally one would like to simply specify the problem and have the GA magically produce a neural network based solution. However, much work remains to be done in the future to enhance the GA to this level of capability.

Appendix A

Effects of Genetic Parameters on Population Diversity

To better understand the relationship between population size, mutation rate and diversity some analytical experiments were performed. Initially the analysis is performed in the absence of selective pressure. To simplify the analysis we study what happens to the diversity of just one bit location of the whole population. We begin by defining the frequency, p, of the population as:

$$p = \frac{W}{N}$$

where W is the number of specimens that have a bit value of 1 and N is the total number of specimen in the population. We let q be the frequency of specimens that have a bit value of 0. Thus, $q = 1 - p$. We can now define a simple measure for the diversity of the population based on the frequency of the population as follows:

$$d(p) = \frac{1}{2} - \left|p - \frac{1}{2}\right|$$

This measure of diversity ranges between 0 and 1/2 with 1/2 representing the maximum level of diversity and 0 representing the fact that all the specimen are identical.

We now determine the probability, p_c, of creating a child specimen with a bit value of 1. We assume that two parents are picked randomly (because we are not applying any selective pressure) and that uniform crossover is used to create the child. For now we consider the case where no mutation is applied after creating the child. The probability of the child having a bit value of 1 is:

$$p_c = p^2 + \frac{1}{2}p(1-p) + \frac{1}{2}(1-p)p$$

This simply reduces to $p_c = p$. Thus, the probability of the child specimen having a bit value of 1 using random selection and uniform crossover is the same as the probability of selecting a specimen from the current population with a bit value of 1. This simplifies the process of creating the next generation of specimen to just making N selections from the current population. Given that the current frequency of the population is p_n we can find the probability of the frequency being p_{n+1} in the next generation as follows:

$$P\langle p_{n+1}|p_n\rangle = p_n^{Np_{n+1}}(1-p_n)^{N(1-p_{n+1})}\binom{N}{Np_{n+1}}$$

For a population of size N, $P\langle p_{n+1}|p_n\rangle$ can be used to define an $(N+1)$x$(N+1)$ matrix M such that each element is given by:

$$m_{ij} = P\left(\frac{i}{N}\middle|\frac{j}{N}\right) \qquad i, j = 0, 1 \ldots N$$

If the current probability distribution for the frequency of specimen is given by g_0, then after k generation the probability distribution of the frequencies can be found by:

$$g_k = \left[wM^k \left[g_0 \right]_D \right]^t$$

g_0 and g_k are $N+1$ element column vectors where the ith element is the probability of the frequency being $\frac{i}{N}$. $\left[g_0 \right]_D$ is a $(N+1)$x$(N+1)$ diagonal matrix with elements of g_0 along the diagonal. w is a $N+1$ element row vector where each element is 1. g_0 will be defined as follows so that initially the population has a high level of diversity.

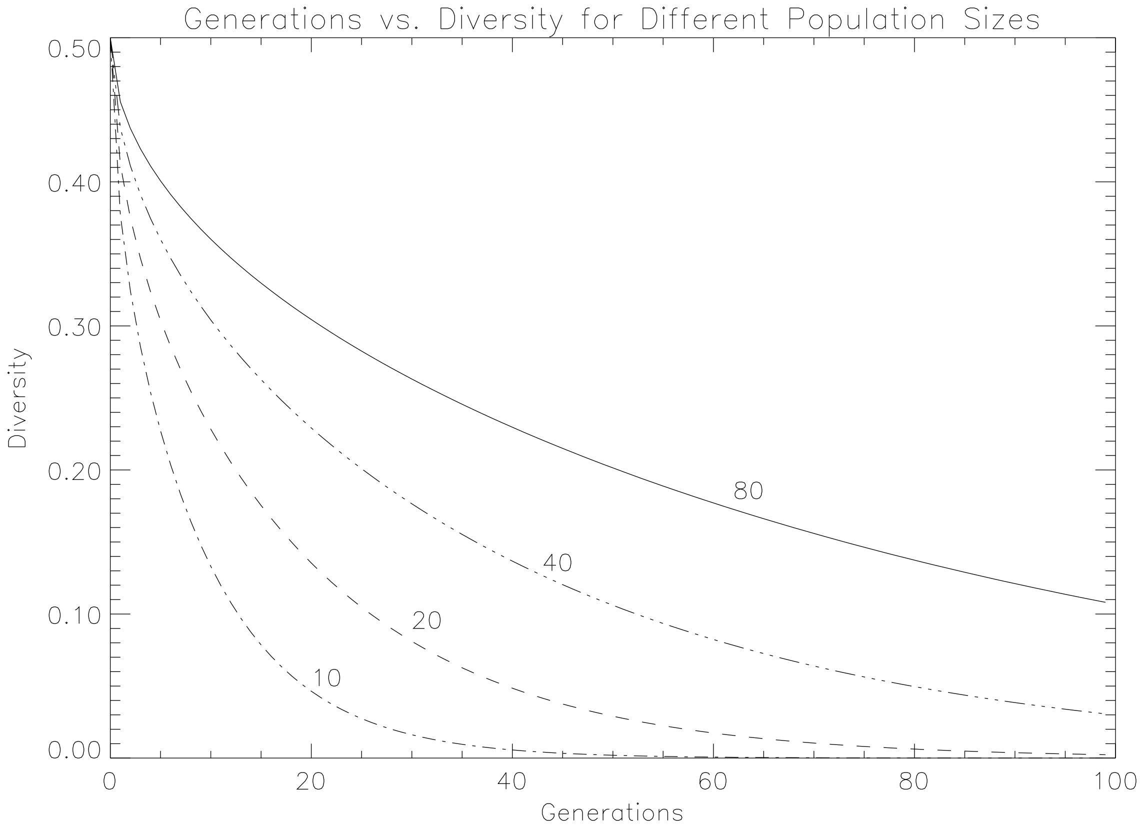
$$g_{0i} = 0 \qquad i = 0, 1 \ldots N, i \neq j$$

$$g_{0j} = \begin{cases} 1 & j = \frac{N}{2} & \text{if } N \text{ is even} \\ \frac{1}{2} & j = \frac{N-1}{2}, \frac{N+1}{2} & \text{if } N \text{ is odd} \end{cases}$$

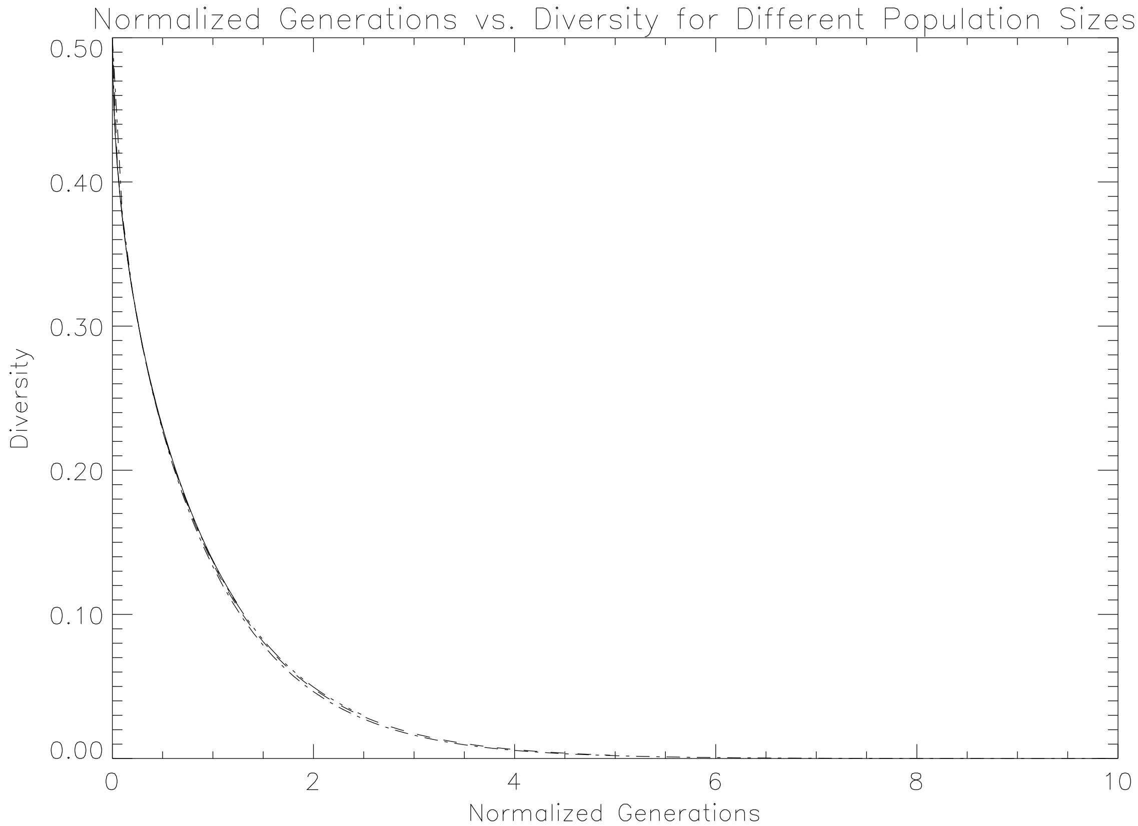
The diversity of the population after k generations, given that the initial probability distribution is g_0, can be found as follows:

$$D(k) = \sum_{i=0}^{N} g_{ki} d(i/N)$$

Using the above equation we can now generate graphs of time (in generations) vs. diversity for populations of different sizes. The following figures shows such graphs for populations of size 10, 20, 40, and 80.



The diversity decreases much faster for smaller sized populations than for larger populations. In fact it seems that the number of generations needed for the diversity to decrease to a certain level is directly proportional to the population size. This is confirmed by the following graph which shows normalized generations (generations / population size) verses diversity. The lines for all population sizes in this graph are identical.



This graph shows that regardless of population size the diversity will go to zero in about $6N$ generations or equivalently $6N^2$ specimen evaluations if mutation is not used. This loss of diversity occurs even in the absence of selective pressure and is referred to as genetic drift in population biology.

We now introduce the effects of mutation and change the way in which we construct the probability distribution matrix M. The rate of mutation is defined to be the probability μ with which any bit in the newly created specimen is toggled. The probability of the child specimen having a bit value of 1 when mutation is used is given by:

$$p_c = p + \mu q - \mu p$$

Which simplifies to:

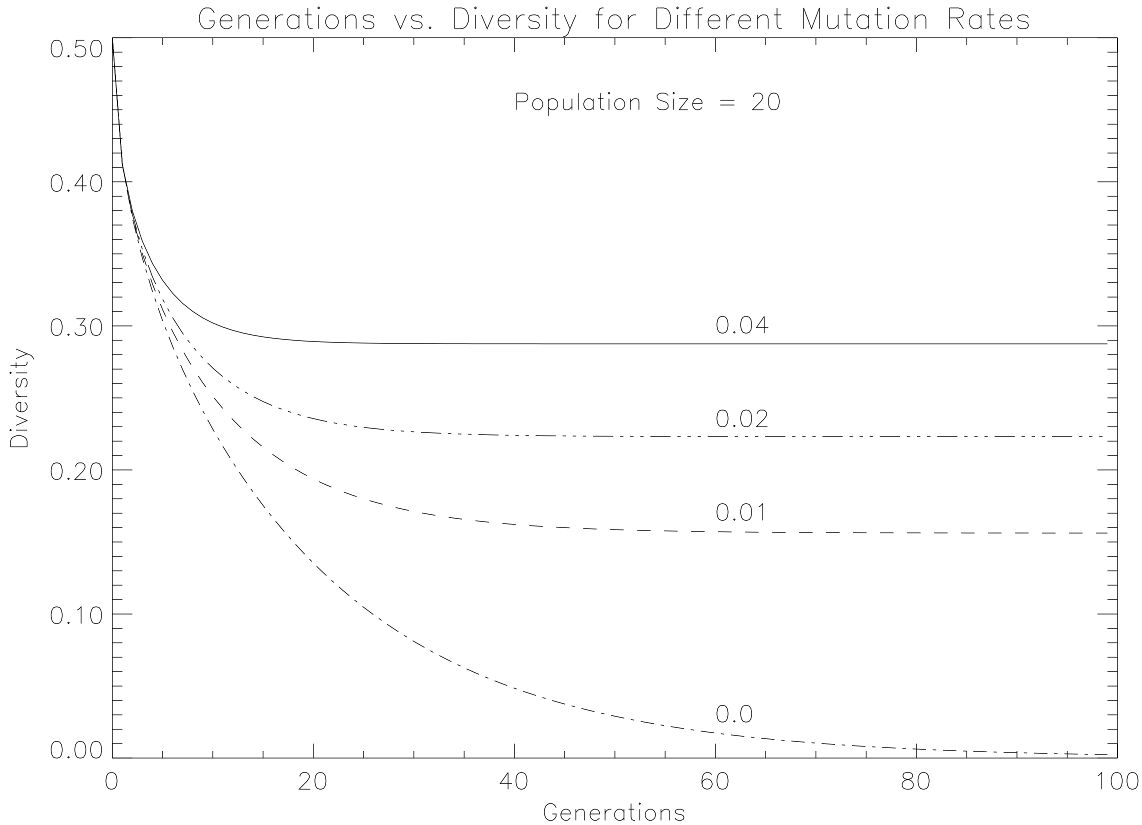
$$p_c = p - 2p\mu + \mu$$

Given that the current frequency of the population is p_n we can find the probability of the frequency being p_{n+1} in the next generation as follows:

$$P\langle p_{n+1} | p_n \rangle = (p_n - 2p_n\mu + \mu)^{Np_{n+1}} (1 - p_n + 2p_\mu\mu - \mu)^{N(1 - p_{n+1})} \binom{N}{Np_{n+1}}$$

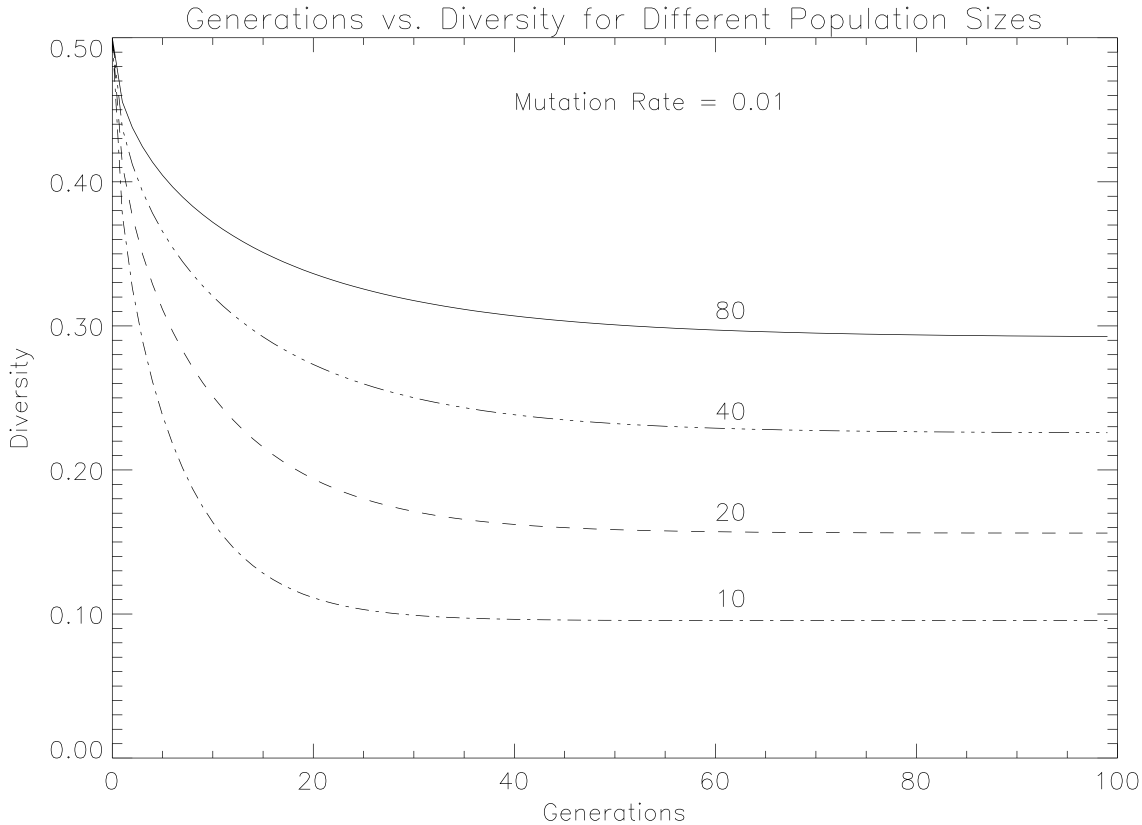
This modified formulation for the probability distribution can now be used to construct the M matrix and compute diversity as a function of generations as before.

The following graph shows the rate at which diversity decreases for various mutation rates using a population size of 20 specimen.



From this graph it is apparent that adding mutation causes the diversity to decrease more slowly and finally reach a minimum level. The diversity does not decrease below this minimum level regardless of the number of generations. Different mutation rates result in different minimum levels of diversity with larger mutation rates leveling off at higher levels of diversity. It is also apparent from this graph that even very small mutation rates have a significant effect on the final level of diversity attained.

The following graph shows how the same mutation rate effects the diversity of different populations sizes.



The same mutation rate effects larger populations more than the smaller ones and causes the larger populations to reach a much higher final level of diversity than smaller populations.

(This raises the question that given the population size and the mutation rate can one compute the diversity at which the population will settle.)

The effects of selection are very difficult to model analytically. However, we can make some simplifying assumptions and try to approximate the effects of selection on diversity. Introducing selection will generally have the effect of increasing the rate at which diversity decreases and possibly causing the final diversity level to be lower. The effects of selection are produced by two completely independent factors. First selection causes a reduction in the effective size of the population. This is simply caused by the fact that not all specimen are selected with an equal probability. Secondly if there is a tendency for a particular bit value to be favored at a bit location by the fitness function, selection will in addition cause the effective frequency of the population to change. The effective frequency may increase or decrease depending on what bit value is favored at the location. Thus, the total effects of selection at a particular bit location depends not only on the parent selection policy used, but also on what bit value is favored for that location by the fitness function.

For any given population the fitness values associated with the specimens defines a ranking such that the specimen can be ordered based on their fitness rank. A parent selection policy can be modeled as probabilities associated with selecting a specimen with a particular rank as the parent. Let ρ_i be the probability of selecting a specimen with rank i as the parent. For rank based selection policies such as tournament selection these probabilities will be constant through out the course of the GA run. For purely fitness proportional selection policies such as roulette wheel selection these probabilities will vary from one generation to the next. In this analysis we consider only rank based selection policies. If uniform crossover is used than the probability that the bit-value of the child will come from a specimen with rank i will be the same as the probability of that specimen being chosen as one of the parents. For inclusive binary tournament selection, with the same specimen possibly being selected more than once, the probabilities are given by:

$$\rho_i = \frac{2(N - i + 1) - 1}{N^2} \qquad i = 1 \ldots N$$

Note that smaller values of i indicate a higher rank.

The unequal probability of selecting specimen as parents results in reducing the effective population size. This is true regardless of weather or not a particular bit-value is being favored for that location. Since our model for computing diversity is based on all specimen in the population being selected with an equal probability (random selection) we must find a new probability ρ_s based on the probabilities ρ_i such that all specimen in the new effective population of size N_s are selected with the probability ρ_s. Also by definition $N_s \rho_s = 1$. We can find the constant probability ρ_s by taking a weighted average of the probabilities ρ_i. Each probability is weighted by how often it is used. Thus, each probability is weighted by itself so that:

$$\rho_s = \sum_{i=1}^{N} \rho_i^2$$

We can use ρ_s to find that the new effective population size N_s is given by:

$$N_s = \frac{1}{\sum_{i=1}^{N} \rho_i^2}$$

If there is a correlation between the bit value of a specimen and its rank then selection also has the effect of increasing or decreasing the effective frequency of the population. To determine the effects of selection on the frequency of the population we must first measure the correlation between a specimens bit value and its rank in the population. The correlation between a specimen having a particular bit value and the specimens rank in the population can be converted to a probability of the particular bit-value occurring at a given rank. Let λ_i be the probability of the bit value 1 occurring at rank location i. Such a set of probabilities can be computed by using k sets with N specimen in each set. The sets must be maximally diverse so that both bit-values are represented equally. The specimens in each set can be ranked based on fitness and the sets combined to produce a histogram h_i of how often the bit-value 1 occurs at rank i. This histogram can be divided by $\frac{kN}{2}$ to find the set of probabilities λ_i associated with the ranks. These probabilities are only representative of the case when both bit-values are represented equally and must be computed for other cases as discussed later. Also these probabilities are likely to change for most bit locations during the course of the GA run. But for most significant or least significant bit locations these probabilities may remain constant. For least significant bits the probabilities should be constant and equal to about $\frac{1}{N}$. For most significant bits the probabilities will increase with increasing rank if the value 1 is favored for that location or decrease if the value 0 is favored.

We now define the correlation probabilities more generally to incorporate unequal representation of the bit-values. Let $h_i^1(p)$ be the histogram produced by ranking k sets of N specimen and counting the number of specimen at rank i that have a bit-value of 1. The sets are chosen such that the frequency of specimen in each set is p. Note that $p = \frac{1}{kN}\sum_{i=1}^{N} h_i^1(p)$. Also the sum at each rank location for a histogram produced for a bit-value of 1 and the histogram produced for a bit-value of 0 should equal k. Thus, $h_i^0(p) = k - h_i^1(p)$.

We define the probability of the bit value 1 occurring at rank location i more generally as:

$$\lambda_i^1(p) = \frac{h_i^1(p)}{\sum_{i=1}^{N} h_i^1(p)}$$

This simplifies to:

$$\lambda_i^1(p) = \frac{h_i^1(p)}{kNp}$$

Similarly we define

$$\lambda_i^0(p) = \frac{h_i^0(p)}{\sum_{i=1}^{N} h_i^0(p)}$$

Since

$$\sum_{i=1}^{N} h_i^0(p) = (1-p)kN$$

This simplifies to:

$$\lambda_i^0(p) = \frac{1 - pN\lambda_i^1(p)}{(1-p)N}$$

If we are only given $h_i^1(1/2)$ we can compute $h_i^1(p)$ using the following:

$$h_i^1(p) = (h_i^1(1/2) - \frac{k}{2})4p(1-p) + kp$$

This equation was constructed so that $h_i^1(0) = 0$, $h_i^1(1) = k$ and $h_i^1(1/2) = h_i^1(1/2)$. This equation can be modified by making a substitution so that we can find $\lambda_i^1(p)$ given $\lambda_i^1(1/2)$.

$$kNp\lambda_i^1(p) = (\frac{kN}{2}\lambda_i^1(1/2) - \frac{k}{2})4p(1-p) + kp$$

This equation simplifies to give:

$$\lambda_i^1(p) = (\lambda_i^1(1/2) - \frac{1}{N})\, 2\,(1-p) + \frac{1}{N}$$

A similar equation can be used to find $\lambda_i^0(p)$ from $\lambda_i^0(1/2)$. Since the final diversity computed should be the same weather we use $\lambda_i^1(p)$ or $\lambda_i^0(p)$, we will no longer refer to the specific bit-value and simply specify $\lambda_i(p)$.

Our original model for computing diversity is based on an equal probability of any bit-value having a particular rank, thus we must compute a new constant probability $\lambda_s(p)$ from the probabilities $\lambda_i(p)$. This constant probability will have a new associated effective population size $N_\lambda(p)$ which can be used to find the effective frequency of the population due to selection using:

$$p_s = \frac{pN}{N_\lambda(p)}$$

We can find the probability $\lambda_s(p)$ by taking a weighted average of the probabilities $\lambda_i(p)$. Each probability is weighted by how often it is used so that:

$$\lambda_s(p) = \sum_{i=1}^{N} \lambda_i(p)\,\rho_i$$

By definition $N_\lambda(p)\,\lambda_s(p) = 1$. This can be used to combine the above equations to find that:

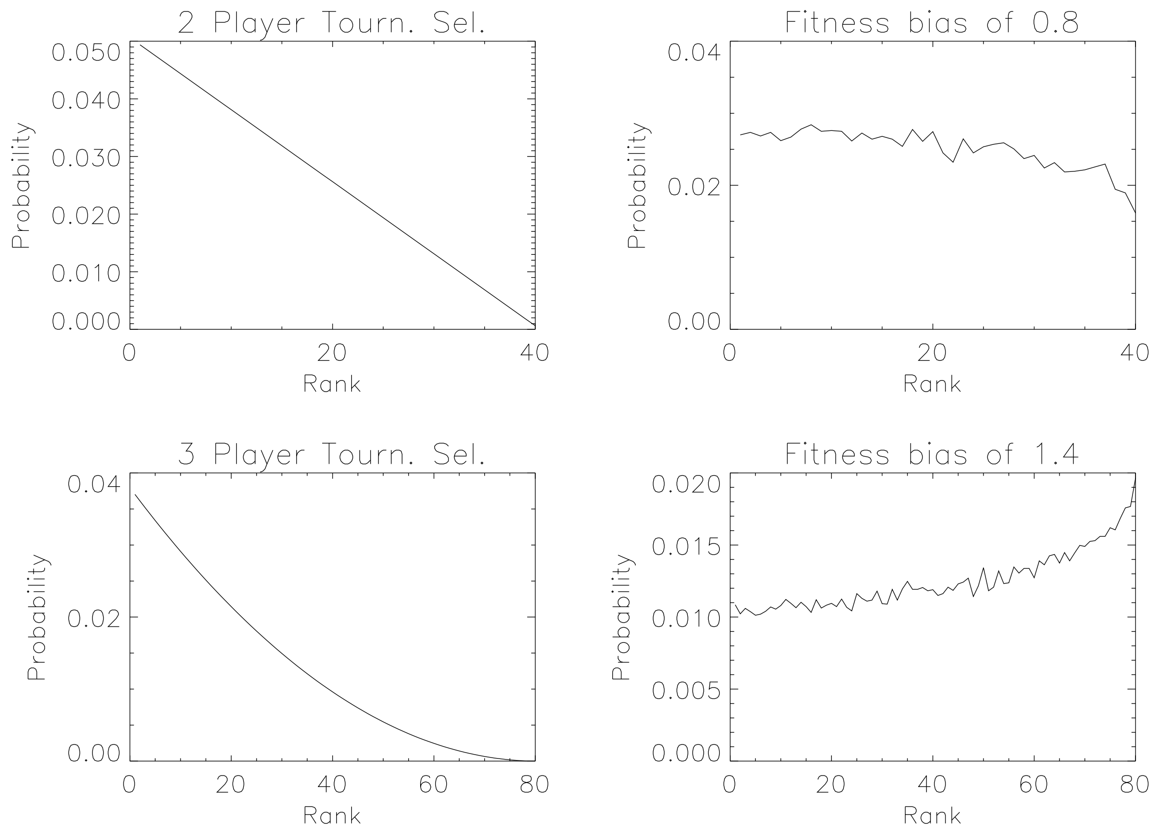
$$p_s = pN\sum_{i=1}^{N}\lambda_i(p)\,\rho_i$$

Note that if all values of ρ_i are constant and equal to $\frac{1}{N}$ as is the case when the selection policy is random selection, the above equation reduces to $p_s = p$. This correctly implies that even if the fitness function favored a particular bit value, the effective frequency of the population does not change if random selection is used. Likewise if a nonrandom selection policy is used but the fitness function does not favor a particular bit-value the effective frequency does not change.

The total effect of selection is to change the effective population size to N_s and to change the effective frequency of the population to p_s. N_s is computed from the probabilities ρ_i due to the selection policy and p_s is computed from both ρ_i and $\lambda_i(1/2)$. Thus, the effective frequency is sensitive to both the selection policy and the fitness function. The effective values can be used in the construction of the M matrix to incorporate the effects of selection. If N_s is not a whole number it will have to be rounded in order to construct the M matrix. In such cases the computed diversity will only be a close approximations to the true diversity.

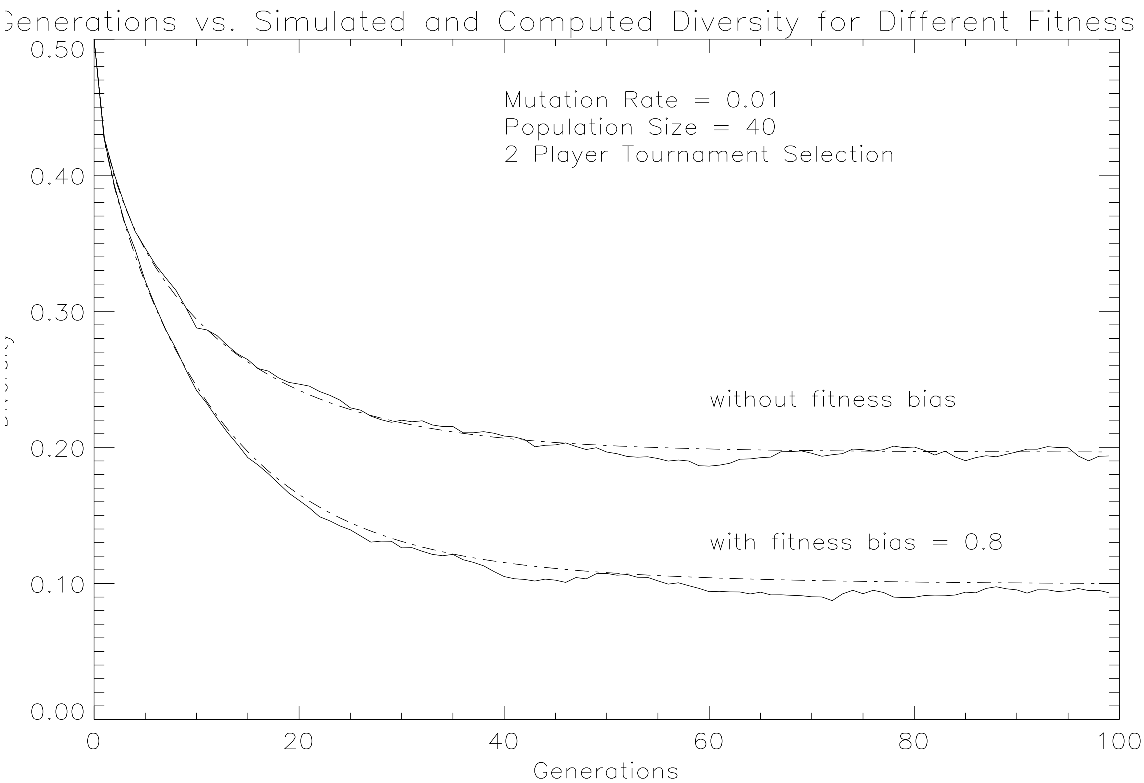
The above formulation for computing diversity was verified against simulation results. Two sets of two experiments were performed. The first experiment in the set used a fitness function which did not favor a particular bit-value. The second experiment in the set used a fitness function that was biased towards a particular bit-value. The first set of experiments used a population size of 40, a mutation rate of 0.01, and binary inclusive tournament selection. The second set of experiments used a population size of 80, a mutation rate of 0.01, and 3 player inclusive tournament selection. A uniform random number in the range [0, 1) was assigned to each specimen as its fitness and used by the selection policy in the first experiment of the sets. In the second experiments the random fitness was assigned as $f = u$ if the bit-value was 0 and $f = u^b$ if the bit-value was 1. u is a uniform random number in the range [0, 1) and b is the fitness bias. Thus, f is also a random number in the range [0, 1), but its distribution is skewed if the bit-value is 1 so that the fitness will tend to be greater or less depending on the value of b. In the second experiment of the first set b was set to 0.8 to favor a bit-value of 1 and was set to 1.4 to not favor a bit-value of 1 in the second experiment of the second set. 1000 trials were run for each experiment and the diversities for all trials were averaged together.

In order to theoretically compute the diversities for each of the four experiments, we need the selection and fitness correlation probabilities for the experiment. The probabilities for selection based on rank can be calculated theoretically knowing the selection policy. However, the probability of a particular bit-value occurring at a specific rank must be found from a simulation. This was done using the procedure described earlier for producing a histogram (using k=1000) and calculating $\lambda_i^1(1/2)$ from it. The following graphs show ρ_i and $\lambda_i^1(1/2)$ for the second experiments of the two sets.

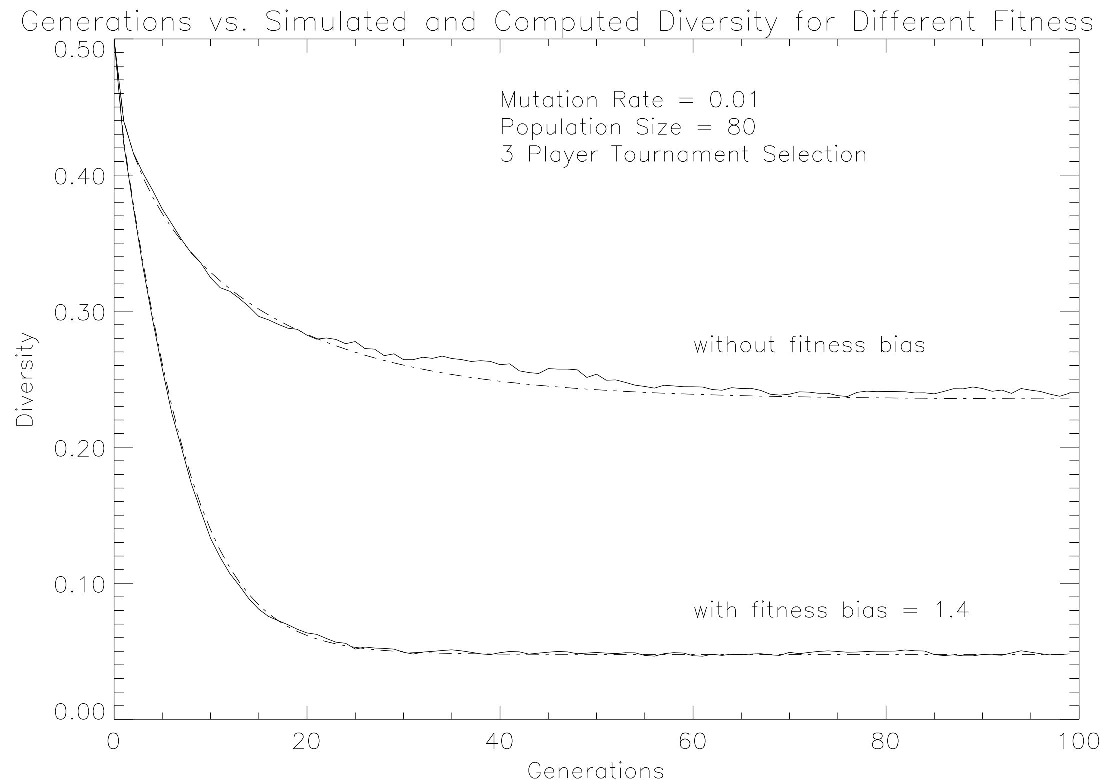


The top graphs are for the second experiment in the first set and the bottom graph is for the second experiment in the second set. The graphs for selection policy show the probability of a specimen of a particular rank being selected as a parent. Note that lower numbers indicate a higher rank. The graphs for fitness bias indicate the probability of a specimen with a bit-value of 1 occurring at a particular rank when the population contains an equal representation of both bit-values.

The following graph show the results of simulated and computed diversities for the two experiments in the first set when the population size was 40.



The solid lines are the results of the simulations, while the smoother dotted lines are the results of the computations. The results for the second set of experiments are shown in the following graph.



The results of the computed and simulated diversities for both sets of experiments match very closely, verifying the correctness of the computational model. The computed diversities were also calculated using $\lambda_i^0(1/2)$ (which was computed from $\lambda_i^1(1/2)$) and as expected were found to give the same results.

Conclusion:

There is a strong relationship between population size, mutation rate and selection on the diversity of a population. Increasing the population size or mutation rate increases the final level of diversity in a nonlinear way.

Increasing selective pressure decreases the final level of diversity. The final level of diversity for a given population size, mutation rate and selective pressure can be computed in most cases using an iterative formulation described earlier. This theoretical computation will be useful in determining weather the level of diversity, when the fitness function does not favor a particular bit, would be high enough to ensure against saturation for the population size and mutation rate being used. If the final level of diversity drops below $1/N$ for this case then the GA will certainly experience premature convergence. Also a computed measure of diversity for the case when the fitness function does not favor a particular bit provides a mean for determining an upper bound on the rate at which the sampling space of new specimen decreases and an upper bound on the final sampling space. The upper bound on the fraction of the solution space being sampled at generation k is given by $(D(k) + 1/2)^B$, where $D(k)$ is the computed diversity and B is the length of a bit string used to represent a potential solution (specimen).

Bibliography

Angeline, P., Gregory, S., & Jordan, P. (1994). *An evolutionary algorithm that constructs recurrent neural networks.* IEEE Transactions on Neural Networks, **5**, 54-65.

Beer, R., & Gallagher, J. (1992). *Evolving dynamical neural networks for adaptive behavior.* Adaptive Behavior, **1**, 91-122.

Bornholdt S., & Graudenz D. (1992). *General asymmetric neural networks and structure design by genetic algorithms.* Neural Networks, **5**, 327-334.

Cybenko, G. (1989). *Approximation by superposition of a sigmoidal function*. Mathematical Control Signals Systems, **2**, 303-314.

Fogel, D., Fogel, L., & Porto, V. (1990). *Evolving neural networks.* Biological Cybernetics, **63**, 487-493.

Fogel, J., Owens J., & Walsh, J. (1966). *Arificial Intelligence through Simulated Evolution*. New York: John Wiley.

Funahashi, K. (1989). *On the approximate realization of continuous mappings by neural networks*. Neural Networks, **2**, 183-192.

Goldberg, D. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley Publishing Company, Inc.

Goldberg, D., Deb, K., & Korb, B. (1991). *Don't worry, be messy.* Proceedings of the Fourth International Conference on Genetic Algorithms, 24-30. La Jolla, CA; Morgan Kaufmann.

Hinton, G., & Nowlan, S. (1989). *How learning can guide evolution*. Complex Systems, **1**, 495-502.

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

Hornik, K., Stinchcombe, M., & White, H. (1989). *Multilayer feedforward networks are universal approximators*. Neural Networks, **2**, 359-366.

Kitano, H. (1990). *Designing neural networks using genetic algorithms with graph generation system.* Complex Systems, **4**, 461-476.

Kuhn, G. (1987). *A first look at phonetic discrimination using a connectionist network with recurrent links.* Technical Report, (SCIMP Working Paper No. 4/87), Communications Research Division, Institute for Defense Analysis, Princeton, NJ.

Montana, D. & Davis, L. (1989). *Training feedforward neural networks using genetic algorithms.* Proceedings of the Eleventh International Joint Conference on Artificial Intelligence IJCAI-89, **1**, 762-767.

Mozer, M. (1988). *A focused back-propagation algorithm for temporal pattern recognition.* Technical Report, Departments of Psychology and Computer Science, University of Toronto, Toronto.

Pearlmutter, B. (1989). *Learning state space trajectories in recurrent neural networks.* Neural Computation, **1**, 263-269.

Peterson, C. & Anderson, J. (1987). *A mean field theory learning algorithm for neural networks.* Complex Systems, **1**.

Pineda, F. (1988). *Dynamics and architecture for neural computation.* Journal of Complexity, **4**, 216-245.

Robinson, A., & Fallside, F. (1987). *The utility driven dynamic error propagation network.* Technical Report CUED/F-INFENG/TR.1, Engineering Department, Cambridge University, Cambridge, England.

Rumelhart, D., & McClelland, J. (1986). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition: Foundations*, volume 1. MIT Press, Cambridge, MA.

Spears, W. & DeJong, K. A. (1990). *An Analysis of Multi-point Crossover.* Proc. 3rd Int'l Conference on Genetic Algorithms. Morgan Kaufman Publishing.

Van den Bout, D. & Miller, T. (1989). *Improving the Performance of the Hopfield-Tank Neural Network Through Normalization and Annealing.* Biological Cybernetics, **62**, 129.

Venugopal, K., Pandya, A., Sudhakar, R. (1994). *A recurrent neural network controller and learning algorithm for the on-line learning control of autonomous underwater vehicles.* Neural Networks, **7**, 833-846.

Whitley, D., Starkweather, T., & Bogart, C. (1990). *Genetic algorithms and neural networks - optimizing connections and connectivity.* Parallel Computing, **14**, 347-361.

Williams, R., & Zipser, D. (1989). *A learning algorithm for continually running fully recurrent neural networks.* Neural Computation, **1**, 270-277.

Zeng, Z., Goodman, R., & Smyth, P. (1993). *Learning finite state machines with self-clustering recurrent networks.* Neural Computation, **5**, 976-990.