The Physics of Intelligence: Genetic Programming as a Case Study

James Robert LePage

University of Tennessee - Knoxville

Follow this and additional works at: https://trace.tennessee.edu/utk_chanhonoproj

Recommended Citation
https://trace.tennessee.edu/utk_chanhonoproj/664
UNIVERSITY HONORS PROGRAM

SENIOR PROJECT - APPROVAL

Name: James Robert LePage, Jr.

College: Arts & Sciences
Department: Physics

Faculty Mentor: Dr. Soren P. Sorensen

PROJECT TITLE: The Physics of Intelligence: Genetic Programming as a Case Study

I have reviewed this completed senior honors thesis with this student and certify that it is a project commensurate with honors level undergraduate research in this field.

Signed: Soren P. Sorensen, Faculty Mentor

Date: 5/8/03

Comments (Optional):

James worked with me on this project during all of the academic year of 2000-2001. He did a great job of defining the project, implementing it, and in the end writing a well written research report. This work clearly fulfills the requirements for a senior project in the University Honors Program.
The Physics of Intelligence:  
Genetic Programming as a Case Study  

Brian Irick  
James LePage  

May 21, 2001  

Supervisor: Dr. Soren P. Sorensen
# Table of Contents

I. Introduction  

II. Design and Construction of Global System  

III. Systemic Evolution  

IV. Observations and Hypotheses  

V. Future Objectives  

VI. Figures
I. Introduction

Here our intent is to briefly summarize the entire project and outline our hypothesis. As the title of this project implies, we are interested in the fundamental principles underlying cognition. Our philosophy is to use a broad definition of Physics; it is a discipline that studies the fundamental properties of a system. When confronted with a system as complex as the human brain we must work hard to simplify it. We can model a complex system in an abstract way by extracting what we believe to be the most essential components.

For the purposes of this study, we are assuming that the most essential components of the brain are individual neurons. However, even a single neuron possesses incredibly complex biological machinery. As Physicists, we will discard much of the neuron’s inherent properties in favor of a “node.” A node simply receives input from other nodes, to which it multiplies a weight and adds a bias. The resulting output is then passed on to various other nodes throughout the system.

Now arises the problem of how to pass input into the system and then receive a single output. An easy way of solving this problem is to create “layers” of nodes that converge to a single output node. This structure is illustrated in Figure 1.1. Now let us assume that there is a certain task we wish the system to perform. In this study we consider the simple game of Tic-Tac-Toe. A certain move is passed as input into the network. This information travels through each layer, passing through multiple nodes. The final output is then interpreted to be a “rating” of the move. Since we are working with such a simple system, it is very obvious what variables will differentiate a good player from a poor one. The only variables to consider are the total number of nodes, their weights and biases, and their connectivity.

The set of all weights of a player’s neural network, along with a maximization function for the output of the neural net, completely characterizes what we refer to as their “evaluation function.” We currently have total connectivity, meaning that each node connects to every node in the next layer. So we have reduced the problem of characterizing a player’s ability to a finite set of numbers: Specifically, the number of layers, number of nodes per layer, and weights. We will refer to the set of optimal nodal weights specific to our task as the “solution space, $S$.” Clearly, optimizing a task-specific set of weights and biases manually is a virtually impossible mathematical problem.

At this point we turn to genetic programming for assistance. First, a population of players is created with random weights and biases. Throughout the study our choice in employing random elements does not reflect the belief that nature is “random.” Instead, we use randomness to create initial conditions, or when
we believe a process to depend on such a large number of variables that it is impossible to directly model with a simple system. For the task of playing Tic-Tac-Toe a tournament is used to rank players, who are then mutated and bred. Using the principles of natural selection we are able to weed out the poor players while retaining the good ones. We can now study various aspects of our artificial system, such as the number of generations required to obtain a player of a given strength. Our global hypothesis is that the strength of a system is proportional to the total number of nodes. In other words, as the number of nodes in a network increases, the system’s ability to perform a task grows. Additionally, its rate of learning may also increase. We believe that it is possible to model this growth as a function of predefined variables.

II. Design and Construction of System Environment

As with the natural world, the system consists of many players within a population which develop through interaction. In our system, this interaction results in genetic variation through the processes of crossover and mutation. We are concerned with optimizing the amount of genetic variation to efficiently search the solution space. Since the overall goal of genetic programming is to obtain improvement in ability, the large majority of our research dealt with understanding the intricacies of breeding between two parents. Once we believed that we understood the principles by which this occurs, we determined that "inbreeding," or breeding restricted to two parents is largely unsuccessful. With larger genetic variation, one can more quickly search the solution space, an optimal set of weights. We suspected, and it did become evident, that with just two parents improvement occurs, but eventually levels off. We interpret this leveling off as a point at which there is no more significant genetic variation. At this point we believe that further improvement can only be obtained through mutation, which is a very slow process. Thus, by introducing a large initial population, we hope to be able to extend the abilities of the players beyond this "leveling off" point.

Neural Networks are at the center of our project, as they comprise the mechanics of a node's evaluation function. During our investigation we have found them to be fairly easy to understand, but very difficult to program. For this reason we opted to use the MATLAB programming environment, which allows for the simple creation and usage of neural networks. We begin by reviewing the structure of a neural net.
A diagram representing an individual neuron is shown in Figure 2.1. An input is fed into the neuron. The input is then multiplied by the weight associated with that input through a dot product. Optionally, an associated bias can then be added to this value. We have chosen to use zero bias in order to hold a currently unnecessary variable constant. This final value is then the output of the neuron. Transfer functions are also frequently added into a neuron to further modify its output. Transfer functions are advantageous in that they allow one to keep the output from the neurons within a certain range. A “pure linear” transfer function simply passes the value from the neural net to the output without altering the value (Figure 2.2). In a more concise drawing we again represent the same neuron in Figure 2.3. Our neural nets have 9 input sources. We illustrate a multiple input, single neuron in Figure 2.4. At this point we will refrain from including biases in further diagrams since they were not employed in our study. Next, a multiple input, multiple neuron, pure linear function is illustrated in Figure 2.5. This neural net consists of 3 inputs, 2 neurons, and 2 outputs. It would be helpful to point out at this stage that the number of outputs is the same as the number of neurons in the terminal layer. Finally, a multilayer, multiple neuron, multiple input neural net is shown in Figure 2.6.

In Figure 2.7 we illustrate an abstract single layer neural net having R inputs and S neurons with a pure linear transfer function. Finally, in Figure 2.8 we show a multilayered neural net in abstract form. It is clear that one can mathematically follow the matrix manipulations used throughout the net. In the final neural net there are R inputs, S 1st layer neurons, and T 2nd layer neurons. For a further review of neural net structure, see Chapter 2 of Neural Network Toolbox for MATLAB User's Guide.

For the purposes of this study, we use a two layer neural net. The first layer in the neural net has a variable number of neurons, while the second layer has only one neuron corresponding with one output. By manipulating the number of first layer neurons we can control the size of the solution space. We currently use a Log-Sigmoid Transfer Function in the first layer and a Pure Linear Transfer Function for the second layer. The Log-Sigmoid Transfer Function maps the neuron’s output to a number in the range from 0 to 1, while the Purelin Transfer Function does not alter the neuron’s output. The neural net takes 9 inputs and gives a single output.

The Tic-Tac-Toe board itself is illustrated in Figure 2.9, along with our chosen numbering scheme. Each position in the picture is assigned a number. This number acts as a numerical address for each position. The board is thereby converted into a 9x1 column vector. We then arbitrarily choose to assign the value of 1 if there is an X in a given position, a value of -1 if there
is an 0, and a value of 0 if the position is empty. Let us use Figure 2.10 as an example board vector.

When a given board vector is fed into the neural net, we obtain a single scalar output. We choose to interpret this output number as a "ranking" of the current board position. We then allow the neural net to "imagine" what would happen if it made a move. This is accomplished by assembling a matrix representing the scenario resulting from each possible move. The move matrix for Player "0, or (-1)" corresponding to our example board is seen in Figure 2.11.

Each column corresponds to the 9x1 board vector resulting from a move in each of the 9 board positions. If a mark is already made in a given position, the board vector is copied unchanged (see column 1 of Figure 2.11). We can now see what the board would look like if 0 were to move into each spot. We now address how the neural net handles this matrix. It is natural to ask what happens when we feed a matrix into the neural net, since up until now we have depicted it only accepting column vector input. MATLAB conveniently breaks down the matrix into column vectors and analyzes each individually. Recalling rules of matrix multiplication, one can see that the output is simply a 1x9 row vector. The first element in the row vector corresponds to the ranking of the first input column in the matrix, the second element of the row vector corresponds to the second input vector, and so on. Therefore, for the matrix in Figure 2.11, we might get the output vector shown in Figure 2.12.

Here you can see that the three board vectors that were the same in the above matrix yielded equal output rankings. The values in our neural net do not generally end up as whole integers, but are values between 0 and 1. Now that we have a ranking, the player must then make a decision. This decision, along with the neural net, is called the "evaluation" function. The evaluation function simply finds the highest ranking for an element in the rank vector that corresponds to an empty space, or 0, in the input vector. This board address is passed as the output of the evaluation function. In our scheme, if two elements have the highest ranking, we arbitrarily choose the first one encountered (farthest left in the rank row vector) and ignore the second one. Ideally this choice might be randomized.

From now on we will refer to the weights used in the neural network as the player's "DNA." This suggestive name illustrates the way in which we model nature. These numbers are used to perform breeding and mutation, serving the same purpose as DNA. Since we do not wish to impart any artificial intelligence to our system, our initial DNA must be completely random. Furthermore, the weights should not change unless by breeding or mutation. Our overall goal is to create a large number of weights (population
space \( \phi \) and then manipulate these weights through breeding and mutation until we reach a point in the solution space, \( S \).

We then encounter the practical problem of storing a large set of numbers in such a way that we can easily access and manipulate them. In MATLAB the easiest way of storing numbers in an external file is by using the .wkl spreadsheet format. In this way, we are able to access the file at any time and easily find a given piece of data. Finally, we are able to pause program execution and view the DNA using any spreadsheet application, making for easier debugging and analysis.

To explain the structure of the DNA file itself (named dna.wkl), consider the example given in Figure 2.13. This is an example of DNA for 2 players, each with 3 neurons in the first layer. In reality there is 15 digit precision in each floating point number. The numbers are color coded to aid in illustrating the layout. Blue numbers correspond to weights for the 1st layer of each player. For every neuron a player has, there is an additional row of DNA. In this case, the first 3 rows are for player 1, and the last 3 rows are for player 2. There are 9 columns of blue numbers which correspond to the 9 inputs that the neural net will take from the Tic Tac Toe board. The red numbers are the weights of the second layer for each player. Note that the number of columns of second layer weights (red) is equal to the number of first layer neurons (number of rows) in each player. Also remember that we will always have one neuron in the second layer, so only one row of red numbers is ever necessary. Finally, the green numbers serve no purpose at all and are simply space-fillers, shortening the DNA-creation algorithm. These numbers can be deleted without affecting program execution. It is critical to understand the DNA structure since it is heavily relied upon for breeding and mutation.

It should be evident at this point that our computer players are "static." In other words, a given player will always open with the same move and play in the same manner given a particular board layout. This brings some predictability into game play. Depending on one's outlook, this may or may not be a desirable characteristic. We have maintained this static nature in the players throughout our research to allow for easier analysis. However, this unchanging nature may be eliminated by using a "stochastic element." This element might simply add a random bias to each of the numbers in the output rank vector. By slightly altering this row vector, a player could spontaneously "change his mind" from what he would typically do. Again, this is simply an attempt to model the extremely complex interactions observed in nature with a chaotic element.

The reference group currently plays a dual role in our program. First of all, we must have some way of measuring performance and thereby any increase in intelligence as we
progress through many generations. Since there are typically at least 1,000,000 games in a given trial, it is completely impractical for humans to play the computer to get a subjective feel for how the population is progressing. Having the system play a group of unchanging players gives us an adequate measure of the changes that occur in our developing group throughout many generations. The reference group consists of static players with one layer neural networks having random DNA.

The second role of the reference group is to aid in choosing players to breed, and in determining which children will survive. In monitoring the performance of players against the reference group, we may choose to reward good developments and punish bad ones. Note that the reference group is not meant to be perfect. We have found it important to find the proper skill level in the reference group so that developing players are able to win or make a tie and thereby receive positive feedback. At the same time, they must be challenged enough that we may distinguish stronger players from weaker.

III. Systemic Evolution

In the process of optimizing weights, it is necessary to develop an efficient and effective method for searching the solution space. Efficiency is found to be important for practical reasons. A single game of Tic-Tac-Toe requires approximately 0.2 seconds of CPU time on a Dual Pentium III 850 MHz PC with 512 MB RAM in our operating environment. Multiplying 0.2 seconds by the number of games required for a tournament followed by the number of generations, we find that the time required for a single trial quickly becomes unwieldy.

However, it is clear that effectiveness is necessary regardless of how long the overall process takes. For this reason, developing effective schemes for selection, breeding, and mutation has been our most difficult task. First let us discuss the process of selecting players, or how to choose and rank players. Since it is our intention to model the natural world, we do not wish to "play god" by imparting artificial intelligence in any aspect of this study. Therefore we choose unranked players randomly and others according to rank.

We must then decide how many players to choose when ranking. In mathematical terms, we wish to rank a subset "N" of the entire population, \( \rho \). In this way we avoid ranking the entire set \( \rho \), thereby reducing the total number of games per generation. This also decreases potential redundancy in ranking and provides a more natural model. In nature, sociological encounters never occur between each and every member of a population.
After experimenting with the sizes of $p$ and $N$, we further hypothesize the existence of a "best ratio" \( R = \frac{|N|}{|p|} \) for fastest searching of the solution space. Experimentally, we have found success with ratios of 10/100, 20/200, and 100/1000. However, while a value of \( R = 10\% \) has been found to yield rapid system development, we have observed better overall success when \(|p|\) is large. This demonstrates the necessity for large amounts of genetic variation, which is expected when searching large, higher dimensional spaces such as $S$. In future studies we intend to study the basis for this empirical result, a macroscopic trend relating rate and amount of improvement to $R$.

During the process of our study we have investigated several tournament styles for ranking the subset $N$. While the round robin tournament (Figure 3.1) is obviously the most effective method, it is not the most efficient. Specifically, the quantity $\text{TotalGames} = (\text{TotalPlayers}) - (\text{TotalPlayers} - 1)$ is sometimes prohibitively large. For this reason we have tried employing various cup methods, specifically a standard and modified cup. The standard cup operates according to the scheme shown in Figure 3.2. Failure in player learning led us to chose the Round Robin tournament style. In making this decision, we eliminated another variable in the system. Additionally, for purposes of calculating trial time beforehand, one should note that it is impossible to predetermine the total number of games in a modified cup tournament.

In an effort to provide accurate ranking, our current scheme involves two games per encounter, one in which each player begins play. In such an encounter there are six possible outcomes: (a) Player 1 wins both games; (b) Player 2 wins both games; (c) Each player wins one game; (d) Both games are tied; (e) Player 1 wins once and ties once; (f) Player 2 wins once and ties once. In cases (a) and (b), one can clearly determine the stronger player. For case (e) and (f) it is again simple to make a choice for the stronger player. However, by evaluating wins and losses alone, we cannot make a definite conclusion as to which player is stronger in cases (c) and (d). In our investigation, we began by assuming that both players were of equal strength and randomly chose one to continue in the cup tournament. This inaccuracy was what initially led us to abandon the cup style in favor of round robin. With the implementation of the new breeding criteria described later in this section, one might argue that the cup tournament may yet prove itself effective.

The method used for ranking players is known as the "fitness function." Since judging a player’s fitness based solely on wins and losses is clearly not sufficient either theoretically or experimentally, the final fitness function in this system also takes into consideration the number of "dumb moves" and speed of victory. Dumb moves is defined to be when a block is not made,
leading to the other player's victory, or when an available win is not taken. The speed of victory is simply the number of moves made in leading to a win.

Initially, when the fitness function was written it only took into account when available wins were not taken. The resulting players had reasonably good offense, but were unable to defend themselves against human players. These player deficiencies lead to the introduction of the blocked move component of the fitness function. It was also at this point that the speed of victory feedback was introduced.

The next version of the fitness function took into account all of the aforementioned variables simultaneously in assigning strength. However, in practice we found that learning was reduced with this new fitness function. We came to the conclusion that when too much feedback, or in our case greater than two variables is all combined into one numerical value, information is lost and little training takes place. At this point we made one final change in the fitness function. The final version allows the operator to choose for a given trial whether to provide feedback for blocked moves or victories not taken. Feedback for speed of victory is provided regardless. See Figure 3.3 for an illustration of the fitness function.

In an ideal run, one type of feedback would be provided for a certain number of generations, followed by the second type, leading to a fully trained player. With this type of feedback there is an issue as to whether strategy is being imparted. Speed of victory seems to be a fairly natural way of giving feedback, since time is often critical in ensuring survival in the wild. However, the validity of using "blocked moves" and "victories not taken" feedbacks in the system is debatable. On one hand we were unable to see much improvement without employing such feedback, while on the other hand some knowledge of the game is implicitly given to the players. One could argue that this negative reinforcement is analogous to a predator in nature, although this predator was artificially engineered.

Initially, our breeding algorithm simply involved taking the arithmetic mean of corresponding elements of the parents DNA. In an effort to model nature, we then implemented a more natural method of breeding. Currently, the DNA is spliced into chunks, or "alleles" of variable size. The offspring's DNA is then assembled by selecting each allele randomly from either parent, while maintaining homogeneity. By homogeneity, we mean that allele ordering and location is maintained in the child. Additionally, a tolerance is used to vary the allele size within a given range.

Originally, it was thought that breeding two good players by this method would yield an equally good or better child. However, empirically we found that there was only about a 10% chance of obtaining a child of better or equal strength. For this reason we
employed the method of "brood recombination." In brood recombination, two parents are bred many times, the resulting children are ranked, and only the strongest children are taken on to the following generation. In this way we ensure that breeding is not a destructive process. This method has proven to be very effective, and resulted in the first significant improvement in overall player strength found in this study (Figure 3.4).

The parameters of the breeding function, allele size and tolerance, were kept fairly constant throughout this study. An allele size of 3 and tolerance of 1 was found to be effective. At this point in the study we wish to minimize the number of variables, and therefore have not experimented extensively with varying these quantities. Future studies might further investigate this aspect of breeding. One can see that breeding is a method of searching the solution space by varying a player's DNA. However, the changes that result from breeding are relatively large. These correspond to large jumps through the solution space. It would be advantageous to also have a way of slowly walking through the space, in the event that a player is close to local maximum. This can be accomplished through mutation, or small, random variances in player DNA. One can think of breeding as a "coarse tuner" on a radio dial, while mutation acts as a "fine tuner."

The original mutation algorithm employed in this study did not accurately simulate mutation in nature. A certain percentage of DNA in the total population was randomly selected for mutation. Then random DNA was generated and replaced the original DNA in the selected areas. However, in nature, rather than producing an entirely new piece of DNA, nature simply varies the current DNA by a small amount. In the final mutation algorithm, numbers are increased or decreased a small amount from what they were previously. This variance was typically set to less than 20% in our study. Again, we opted to keep this variable fairly constant throughout the study in order to reduce confusion.

In observing the DNA after many generations of breeding, much of it was clearly identical between players. This was the result of inbreeding between successful players. At this point, we often observed a "flat line" of player strength, in which little progress took place between generations. Mutation seemed to provide a slow escape from these plateau regions. It should also be noted that mutation was found to be extremely destructive when used with large variance.

IV. Observations and Hypotheses

In order to determine how player performance proceeded we decided to monitor how many games were won by the strongest
player in each generation. The number of games won was then plotted against number of generations. We also found that there are several other quantities that are helpful to measure and plot. These include number of games lost, number of cat wins (ties), and a total score. In the current system each win corresponds with 1 point and each tie corresponds with \( \frac{1}{2} \) point.

For future research, it would be helpful to also plot the resulting score arising from the evaluation of the fitness function described in Part III. Monitoring this number is crucial since it decides which players are to breed. This may reveal further insights into our study, and give a more accurate graphical depiction of overall player ability. As of yet we have not explored this method of measuring player strength.

In our initial plots we decided to analyze the improvements of the top two players. We opted for this method in an effort to seek a balance between execution time and reliability of accurately measuring the entire population. In this way we could ensure that improvement was made to more than just the strongest player. There is therefore some assurance that the total population is improving rather than just one select individual. It should not be surprising that the second strongest player is only about a generation behind the strongest player in terms of strength. We interpret this empirical fact to mean that it takes approximately one or two generations for a new development to begin propagating throughout the population.

After implementing the brood recombination technique, but before implementing the additional breeding criteria as discussed in Part III, we typically observe a big jump in performance within a few generations and then a flat region that seemed to last indefinitely. SEE Figure 4.1.

After we began to implement additional breeding criteria, we found that we were able to break out of the plateau regions, albeit slowly. With large populations, there is a lot of scattering within initial generations. However, one may easily see that this scattering quickly begins to converge toward a plateau. This convergence occurs as players breed and the population begins to approach one universal (the strongest) set of DNA. When the first plateau region is reached, almost every member in the population has similar DNA with small variations due to mutation. Because of the large priority assigned to the speed at which the players win, one may hypothesize what the first plateau region corresponds to. The first thing we observe players to develop is what we call a "three move". In other words, one horizontal, vertical, or diagonal win quickly becomes prevalent. By the time the first plateau region is reached the player will almost invariably try to achieve this three win. However, it is fairly easy to foil this strategy, even by randomly dumb reference players. Only due to the additional
breeding criteria are we able to break out of the plateau region. Every time players escape from a plateau region, there is apparently a direct correspondence to a newly formed strategy.

This corresponds with the following if/then logic:

1\textsuperscript{st} Plateau - If: I can make a three win
   Then: Make the three win

2\textsuperscript{nd} Plateau - If: I can not make the three win,
   - Then: Move to another spot where a potential win may be made

3\textsuperscript{rd} Plateau - If: I can not make the potential win
   - Then: Determine whether another potential win may be made

We expect that additional plateaus may be reached. For example:

7\textsuperscript{th} Plateau - If: No win, but tie may be made
   - Then: Make the tie

It should be interesting to note that the best player obtained was in one of the few population sets that lost their genetic variation rather quickly, and slowly worked its way through many plateaus (Figure 4.2). This suggests that "fine" strategy simply isn't developed until many of these plateau regions have occurred. A small population is observed to lose its genetic variance quickly. Such a population that travels through multiple plateaus seems to be significantly stronger than sets with huge initial populations. Large populations lose their genetic variation slowly and do not reach many advantageous plateau regions quickly.

One should also note that plateau regions are not broken instantly when a beneficial mutation or crossover occurs. Rather, only one player will typically develop this new trait, and it then takes additional generations to propagate throughout the population. In other words, there might be an increase in one generation from the plateau region, then a temporary regression, and finally a new, higher plateau. In other words, one may see the new plateau coming even before it completely arrives. This would suggest that we may predict how long it would be before a beneficial trait propagates throughout the entire population. Thus, we make the following claim:

The number of generations required for an improvement to propagate throughout the population after its initial development within one player is equal to $\sqrt{\text{Total Players}}$.
Since our DNA is purely numerical, one would expect to be able to make many mathematical predictions regarding rate of learning. This particular hypothesis is derived from the mathematical property of the "random walk." This theorem states that when making a random journey of \( n \) steps, the final destination will be approximately \( \sqrt{n} \) units away. There is a direct analogy to making a random walk through \( n \) players in a population. More testing is required to verify this hypothesis.

As has been noted, the player intelligence rapidly increases during the first generations and later reaches plateaus. The learning process seems to occur slowly after the plateau regions. We managed to find ways of expediting this process by employing additional breeding criteria. We also suggest that the process may be accelerated by breeding based on one or two criteria at a time rather than all the criteria at once. The weights given to each of the breeding criteria are arbitrary and may not be the most efficient. We may eventually observe that for different given weights, learning occurs more rapidly. This should be investigated in later research.

V. Future Objectives

In the future we hope to further demonstrate the observations and hypotheses mentioned in Part IV. This will obviously entail running many more trials, isolating and studying one variable at a time. At this point we do not feel it necessary to make any major changes in the code. It should be noted that most of the Genetic Programming community favors LISP as a platform. The book *Genetic Programming* by Koza outlines methods of using this string-based programming language.

In a string-based GP setup, the player DNA is arranged as Boolean trees. These trees form the basis for the player's evaluation function. Since the system developed in this study is purely numerical (with the use of a neural net, which is not typical in GP research), we feel that if it proves effective it will be in many ways superior to tree-based forms of DNA in modeling nature. Up until now, genetic programming has mostly been used in the computer science community as a practical method of generating machines that are adept at performing a certain task, with no concern for modeling nature. While tree-based DNA is far easier for a human to read and interpret, numerical evaluation forms a much more natural correspondence to the process of neuron firing. For this reason, any success in using numerical DNA will prove to be a milestone in genetic programming for use in modeling nature.

Currently the GP system developed in this study suffers from a major drawback, the lack of a stochastic element. The players
generated by this system will always make the same move in a given situation. While we believe that humans are chemical machines that act in much the same way, we also acknowledge that in nature humans are never presented with identical situations and for that reason humans rarely behave in a uniform manner. The human experience simply consists of too many variables for any repetition to be probable.

As has been mentioned several times throughout this paper, we choose to model extreme instances of complexity in the human system by using random elements. We therefore hope to introduce a stochastic element that serves to produce a more dynamic player. In this way players will no longer be static, and will behave differently in every game. We must question whether dynamic players will still maintain a high skill level, particularly in a game as simple as Tic-Tac-Toe where it is quite feasible for a human to play “perfectly” each time.

Another flaw of the current GP system in modeling nature is the lack of player development in successive generations. In one sense our use of the word “DNA” is inconvenient, since a player’s DNA currently also serves as their intelligence. In nature an organism’s intelligence can develop independent of their DNA, in the formation of new synapses between neurons. We therefore wish to introduce a method of altering connectivity between nodes within a trial. Rather than only observing a societal increase in ability, this method would also allow individual players to develop.

One final goal in sight for the future is allowing for task independence. It is clear that the players generated in this Tic-Tac-Toe system would be completely inept at playing checkers or chess. In fact, their hardwired inputs and outputs would not even “understand” how to operate in any game other than Tic-Tac-Toe. In the ongoing effort at modeling nature, we must recognize the fact that most organisms are able to perform many tasks. We are constantly seeking to further generalize our software, in an effort to create fully adaptable machine intelligence.

While we currently do not wish to entirely rewrite our current code, it may be useful in the future to have a C++ version. Matlab does not currently support the generation of standalone executables. If we ever wish to move to a distributed network method of operation, it would be advantageous to have standalone executables of our software. Also, for purposes of making our software available to others, we may wish to create a “Numerical GP” C++ class library. Any other program revision would simply involve commenting and cleaning up our current Matlab code.

Let us now address our original global hypothesis: “The strength of a system is proportional to the total number of nodes.” We have concluded through the course of this study that
our original hypothesis was somewhat limited in scope. We have encountered the following variables:

- Global Environment
  - |p|
  - |N|
  - Number of Reference Players
  - Total Generations
  - Number of Neuron Layers and Neurons per Layer
  - Type of Transfer Function

- Crossover
  - Allele Size
  - Allele Tolerance
  - Number of Children Bred per Crossover

- Mutation
  - Mutation Percentage
  - Variance

- Fitness Function
  - Breeding Criteria and Associated Weight

It is clear that many more hypotheses may be formed in dealing with this relatively complex system. Number of neurons is simply one aspect of the overall scheme. We are unprepared at this time to say whether a large number of nodes is necessarily beneficial for all tasks. One might argue that for a game as simple as Tic-Tac-Toe, it is possible to have too many neurons. Therefore, we may wish to revise the overall hypothesis and further investigate the many other facets of this project, some of which are described in Part IV. It is evident in our study that there is simply much more to developing intelligence than one single variable.
VI. Figures

1.1 - Node Layout in Neural Network

2.1 - Individual Neuron
2.2 - Transfer Function

![Transfer Function Diagram]

2.3 - Individual Neuron, Concise Drawing

![Individual Neuron Diagram]

2.4 - Multiple Input, Single Neuron

![Multiple Input Diagram]
2.5 - Multiple Input, Multiple Neuron, Pure Linear Network Consisting of 3 Inputs, 2 Neurons, and 2 Outputs

2.6 - Multilayer, Multiple Neuron, Multiple Input Neural Net
2.7 - Abstract, Single Layer Neural Net With R Inputs and S Neurons With a Pure Linear Transfer Function

2.8 - Multilayered Neural Net in Abstract Form
2.9 - Tic-Tac-Toe Board With Chosen Numbering Scheme

2.10 - Example Board Vector

2.11 - Move Matrix for Player "O, or (-1)" Corresponding With Example Board in Figure 2.10

\[
\begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\
-1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\
0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1
\end{bmatrix}
\]
2.12 - Potentially Resulting 1x9 Output Rank Vector Corresponding With Figures 2.10, 2.11

\[ [1, 3, 4, 9, 1, 5, 3, 2, 1] \]

2.13 - Example DNA File (dna.wk1)

0.950 0.456 0.922 0.410 0.139 0.015 0.846 0.681 0.305 0.151 0.497 0.342
0.231 0.019 0.738 0.894 0.203 0.747 0.525 0.379 0.190
0.607 0.821 0.176 0.058 0.199 0.445 0.203 0.832 0.193
0.486 0.445 0.406 0.353 0.604 0.932 0.672 0.503 0.682 0.860 0.645 0.534
0.891 0.615 0.935 0.813 0.272 0.466 0.838 0.709 0.303
0.762 0.792 0.917 0.010 0.199 0.419 0.020 0.429 0.542

3.1 - Round Robin Tournament
3.2 - Standard Cup Tournament

Choice = 1: Offensive Training
Choice = 2: Defensive Training

if choice == 1
    bredPlayerStrength = bredPlayerStrength + newPlayerStrength + NML;
elseif choice == 0
    bredPlayerStrength = bredPlayerStrength + newPlayerStrength + BM;
end

% BredPlayerStrength - Number of games player wins
% newPlayerStrength - negative number - number of times player makes a dumb move
% NML - number of moves that were left in the game -- essentially how fast the player won
% BM - negative number corresponding to number of winning moves not blocked
3.4 - First Trial Demonstrating Improvement

Parameters:
Total Population = 100
# Randomly Chosen = 10
Total Generations = 50
Neurons in Neural Net = 3
Children Bred = 30
Reference Group Size = 20
Allele Size = 3
Allele Size Variance = 1
Mutation = 5%
Mutation Offset = 0.2
4.1 - Trial Showing Single Plateau (continuation of Figure 3.4)
4.2 - Trial Showing Many Plateau Regions (2 Continuous Graphs)

Figure 1

Player 1 Wins against Reference Group

Cat Wins

Overall Strength of Player 1 Against Reference Group

Player 1 losses against Reference Group
Figure 4.2 Cont’d. (Generations 101-200)