TABLE OF CONTENTS

Computer Science Department June 1990

GENETIC PROGRAMMING: A PARADIGM FOR GENETICALLY BREEDING POPULATIONS OF COMPUTER PROGRAMS TO SOLVE PROBLEMS

John R. Koza
(Koza@Sunburn.Stanford.Edu)

Computer Science Department
Stanford University
Margaret Jacks Hall
Stanford, CA 94305

GENETIC PROGRAMMING: A PARADIGM FOR GENETICALLY BREEDING POPULATIONS OF COMPUTER PROGRAMS TO SOLVE PROBLEMS

John R. Koza
Computer Science Department
Stanford University
Margaret Jacks Hall
Stanford, CA 94305
Koza@Sunburn.Stanford.Edu

ABSTRACT: Many seemingly different problems in artificial intelligence, symbolic processing, and machine learning can be viewed as requiring discovery of a computer program that produces some desired output for particular inputs. When viewed in this way, the process of solving these problems becomes equivalent to searching a space of possible computer programs for a most fit individual computer program. The new "genetic programming" paradigm described herein provides a way to search for this most fit individual computer program. In this new "genetic programming" paradigm, populations of computer programs are genetically bred using the Darwinian principle of survival of the fittest and using a genetic crossover (recombination) operator appropriate for genetically mating computer programs. In this paper, the process of formulating and solving problems using this new paradigm is illustrated using examples from various areas.

Examples come from the areas of machine learning of a function; planning; sequence induction; symbolic function identification (including symbolic regression, empirical discovery, "data to function" symbolic integration, "data to function" symbolic differentiation); solving equations, including differential equations, integral equations, and functional equations); concept formation; automatic programming; pattern recognition, time-optimal control; playing differential pursuer-evader games; neural network design; and finding a game-playing strategy for a discrete game in extensive form.

1. INTRODUCTION AND OVERVIEW

Many seemingly different problems from artificial intelligence, symbolic processing, and machine learning can be viewed as requiring the discovery of a computer program that produces some desired output when presented with particular inputs. Depending on the terminology of the particular field involved, the "computer program" may be called a robotic action plan, an optimal control strategy, a decision tree, an econometric model, a set of state transition equations, the transfer function, a game-playing strategy, or, perhaps more generically, a "composition of functions." Similarly, the "inputs" to the "computer program" may be called sensor values, state variables, independent variables, attributes, or, perhaps merely, "arguments to a function." However, regardless of different terminology used, the underlying common problem is discovery of a computer program that produces some desired output when presented with particular inputs.

The purpose of this paper is to show how to reformulate these seemingly different problems into a common form (i.e. a problem requiring discovery of a computer program) and, then, to describe a single, unified approach for solving problems formulated in this common form.

1.1. EXAMPLES OF PROBLEMS REQUIRING DISCOVERY OF A COMPUTER PROGRAM

The following problems are among the many problems that can be viewed as problems of discovering a computer program that produces some desired output when presented with particular inputs:

- Machine learning of a function requires discovering a composition of functions that can return the correct value of a function after seeing a relatively small number of specific examples associating the value of the function with particular combinations of arguments. The problem of learning the Boolean multiplexer function is an example of machine learning of a function. In this paper, we show how to discover a composition of elementary Boolean functions for the Boolean 11-multiplexer function (Section 4.1).

- Planning in artificial intelligence and robotics requires finding a plan that receives information from sensors about the state of various objects in a system and then uses that information to select a sequence of actions which change the state of the objects in the system. In this paper, we show how to generate a general plan for correctly stacking labeled blocks onto a target tower in a specified order. We then show how to generate a more efficient plan to solve the same problem by including the plan's efficiency as a component of the fitness measure. We also show how to generate a plan allowing an "artificial ant" to traverse a trail with gaps of various types (Section 4.2).

- Symbolic function identification requires finding a function, in symbolic form, that fits a given discrete set of data points. Symbolic function identification problems appear in various forms. The simplest form is sequence induction. Sequence induction requires developing a computational procedure that can generate any arbitrary element in a sequence $S = S_0, S_1, ..., S_j, ...$ after seeing only a relatively small number of specific examples of the values of the sequence. For example, one might easily induce the computational procedure j^2+1 given only the six sequence values 1, 2, 5, 10, 17, 26 for sequence positions j from 0 through 5. On the other hand, it is considerably harder to identify the computational procedure for S = 1, 1, 2, 3, 3, 4, 5, 5, 6, 6, 6, 8, 8, 8, 10, 9, 10, 11, 11, 12. In any case, induction is at the heart of learning and the ability to correctly perform induction is widely accepted as a means for measuring human intelligence. In this paper we demonstrate how to perform sequence induction using the "genetic programming" paradigm and find a correct computational procedure for the Fibonacci and Hofstadter sequences (Section 4.3.1).

- "Symbolic regression" is another form of the problem of symbolic function identification. Symbolic regression requires finding a function, in symbolic form, that fits a given sampling of values of the dependent variable associated with particular given values of the independent variable(s). While conventional linear, quadratic, or higher order polynomial regression requires merely finding the numeric coefficients for a function of a pre-specified functional form, symbolic regression involves finding both the appropriate functional form and the appropriate numeric coefficients for the model. In this paper, we show how to find a composition of mathematical functions and numeric constants that fits a given sampling of data (Section 4.3.2).

- "Empirical discovery" also requires finding a function, in symbolic form, that fits given numeric data points representing some observed system. Finding such an empirical model for a system can also be used in forecasting future values of the state variables of the system. In this paper, we rediscover the well-known non-linear econometric "exchange equation" M= PQ/V relating the time series for the money supply with the price level, gross national product, and velocity of money in an economy (Hallman 1989). We also rediscover Kepler's third law from empirically observed planetary data and discover the functional form of a simple "chaotic" function (Section 4.3.3).

- Symbolic "data to function" integration and symbolic "data to function" differentiation involve finding the integral or derivative in symbolic form, of an unknown curve represented by a sampling of discrete data points Sections 4.3.5. and 4.3.6).

- Solving functional equations involves finding a function that satisfies the given functional equation. In this paper, we show how to solve a differential equation for a function, in symbolic form, that satisfies (or approximately satisfies) the differential equation and its associated initial conditions. We also discover a Taylor power series as the solution to a differential equation. We similarly solve integral equations for the function, in symbolic form, that satisfies (or approximately satisfies) the integral equation. We also solve an inverse problem and other more general functional equations. The special case of solving an equation for numeric (i.e. non-functional) roots is also presented. (Section 4.4).

- Concept formation requires developing a decision tree for classifying any object in a universe into a particular class on the basis of the attributes of that object. In this paper we show how to induce a decision tree to classify objects into classes (Section 4.5).
- Automatic programming requires developing a computer program that can produce a desired output for a given set of inputs. In this paper, we show how to find a computational procedure for solving a given pair of linear equations $a_{11}x_1 + a_{12}x_2 = b_1$ and $a_{21}x_1 + a_{22}x_2 = b_2$ for the real-valued variable x_1; how to find a computational procedure for solving a given quadratic equation $ax^2 + bx + c = 0$ for a complex-valued root x_1; and how to solve certain trigonometric identities. We also show how to solve other computational problems, such as finding the prime numbers (Section 4.6).
- Pattern recognition requires finding a computational procedure that processes a digitized input image to determine whether a particular pattern is present in the input image. In this paper, we show how to do translation-invariant pattern recognition of a simple one-dimensional shape in a linear retina (Section 4.7).
- Time-optimal control requires finding a control strategy that uses the values of the state variables of a system to select a sequence of actions to change the state of the system to a desired new state in minimal time. An example of a time-optimal control problem involves discovering a non-linear "bang bang" control strategy for centering a cart and balancing a broom in minimal time (Section 4.8).
- Game playing requires finding a strategy for a player that specifies what move that player is to make at each point in the game. In this paper, we show how to find the minimax strategy for the pursuer to catch an evader in the differential "game of simple pursuit" played on the plane . We also show how to find the minimax game-playing strategy for a discrete game in extensive form (Section 4.9).
- "Multiple regression" is another form of the problem of symbolic function identification. Multiple regression requires finding a multiple-valued function, in symbolic form, that fits a given sampling of values of dependent variables associated with particular given values of the independent variable(s). The problem of multiple regression illustrates the additional requirement of creating random random individuals that comply with a particular set of syntactic restrictions and then performing structure-preserving crossover on those individuals In this paper, we show how to find the complex multiplication function (Section 4.10.1).
- Neural network design and training involves finding a neural network architecture and associated set of weights so that the neural network can perform a desired task. In particular, in this paper, we show how to discover the appropriate number of layers for the neural net, the number of processing elements in each layer, the connections between processing elements, and the required weights. The problem of neural network design illustrates the additional requirement of creating random random individuals that comply with a particular set of syntactic restrictions and then performing structure-preserving crossover on those individuals (Section 4.10.2).

1.2. SOLVING PROBLEMS REQUIRING DISCOVERY OF A COMPUTER PROGRAM

Problems of the type described above can be expeditiously solved only if the flexibility found in computer programs is available. The flexibility we seek includes the ability to create and execute computer programs whose size, shape, and complexity is not specified in advance, to perform alternative computations conditioned on the outcome of intermediate calculations, to perform computations on variables of many different types, to perform iterations and recursions to achieve the desired result, to define and subsequently use computed values and sub-programs, and to perform all of these operations in a hierarchical way

We claim that the process of solving these problems can be reformulated as a search for a most fit individual computer program in the space of possible computer programs. In particular, the search space is the hyperspace of LISP “symbolic expressions” (called S-expressions) composed of functions and terminals appropriate to the problem domain.

As will be seen, the LISP S-expression which solves each of the problems described above will emerge from a simulated evolutionary process using a new “genetic programming paradigm" using a "hierarchical genetic algorithm."

In each case, this simulated evolutionary process starts with an initial population of randomly generated LISP S-expressions composed of functions and terminals appropriate to the problem domain. The functions may be standard arithmetic operations, standard programming operations, standard mathematical functions, and various domain-specific functions.

A fitness function measures how well each individual LISP S-expression in the population performs in the particular problem environment. In many problems, the fitness is measured by the sum of the distances (taken for all the environmental cases) between the point in the range space (whether Boolean-valued, integer-valued, real-valued, complex-valued, vector-valued, symbolic-valued, or multiple-valued) created by the S-expression for a given set of arguments and the correct point in the range space. The closer this sum is to zero, the better the S-expression.

Predictably, the initial random individual S-expressions has exceedingly poor fitness. Nonetheless, some individuals in the population are somewhat more fit than others. And, in the valley of the blind, the one-eyed man is king.

Then, an algorithm based on the Darwinian model of reproduction and survival of the fittest and genetic recombination is used to create a new population of individuals from the current population of individuals. In particular, a genetic process of sexual reproduction between two parental S-expressions is used to create offspring S-expressions. The two participating parental S-expressions are selected in proportion to fitness. The resulting offspring S-expressions are composed of sub-expressions ("building blocks") from their parents. Then, the new population of offspring (i.e. the new generation) replaces the old population of parents (i.e. the old generation). Then, each individual in the new population is measured with the fitness function and the process is repeated.

At each stage of this highly parallel, locally controlled, and decentralized process, the state of the process will consist only of the current population of individuals. Moreover, the only input to the algorithmic process will be the observed fitness of the individuals in the current population in grappling with the problem environment.

As will be seen, this algorithm will produce populations which, over a period of generations, tend to exhibit increasing average fitness in dealing with their environment, and which, in addition, will tend to robustly (i.e. rapidly and effectively) adapt to changes in the environment.

The solution produced by this algorithm at any given time can be viewed as the entire population of disjunctive alternative solutions (typically with improved overall average fitness as compared to the beginning of the algorithm), or, more commonly, as the single best individual in the population at that time ("winner take all").

The hierarchical character of the computer programs that are produced by the genetic programming paradigm is an important feature of the “genetic programming” paradigm. The results of this genetic programming paradigm process are inherently hierarchical. And, in many cases, the results often contain default hierarchies which solve the problem in a relatively parsimonious way.

The dynamic variability of the computer programs that are developed along the way to a solution is also an important feature of the genetic programming paradigm. In each case, it would be difficult and unnatural to try to specify or restrict the size and shape of the eventual solution in advance.

Moreover, the advance specification or restriction of the size and shape of the solution to a problem narrows the window by which the system views the world and might well preclude finding the solution to the problem.

Another important feature of the genetic programming paradigm is absence of preprocessing of inputs and the fact that the solution is expressed directly in terms of the functions and arguments from the problem domain. This makes the results immediately comprehensible and intelligible in the terms of the problem domain.

Most importantly, the "genetic programming" paradigm is general and provides a single, unified approach to a variety of seemingly different problems in a variety of areas.

2. BACKGROUND ON GENETIC ALGORITHMS

Observing that sexual reproduction in conjunction with Darwinian natural selection based on reproduction and survival of the fittest enables biological species to robustly adapt to their environment, Professor John Holland of the University of Michigan presented the pioneering mathematical formulation of simulated evolution ("genetic algorithms") for fixed-length (typically binary) character strings in *Adaptation in Natural and Artificial Systems* (Holland 1975).

In this work, Holland demonstrated that a wide variety of different problems in adaptive systems are susceptible to reformulation in genetic terms so that they can potentially be solved by a highly parallel mathematical "genetic algorithm" that simulates Darwinian evolutionary processes and naturally occurring genetic operations on chromosomes.

Genetic algorithms superficially seem to process only the particular individual binary character strings actually present in the current population. However, Holland's 1975 work focused attention on the fact that they actually also implicitly process, in parallel, large amounts of useful information concerning unseen Boolean hyperplanes (called schemata) representing numerous additional similar individuals not actually present in the current population. Thus, genetic algorithms have a property of "intrinsic parallelism" which enable them to create individual strings for the new population in such a way that the hyperplanes representing these unseen similar other individuals are all automatically expected to be represented in proportion to the fitness of the hyperplane relative to the average population fitness. Moreover, this additional computation is accomplished without any explicit computation or memory beyond the population itself. As Schaffer (1987) points out, "Since there are very many more than N hyperplanes represented in a population of N strings, this constitutes the only known example of the combinatorial explosion working to advantage instead of disadvantage."

In addition, Holland established that the seemingly unprepossessing genetic operation of crossover in conjunction with the straight forward operation of fitness proportionate reproduction causes the unseen hyperplanes (schemata) to grow (and decay) from generation to generation at rates that are mathematically near optimal. In particular, Holland established that the genetic algorithm is a mathematically near optimal approach to adaptation in the sense that it maximizes overall expected payoff when the adaptive process is viewed as a set of multi-armed slot machine problems for allocating future trials in the search space given currently available information.

Holland's l975 work also highlighted the relative unimportance of mutation in the evolutionary process. In this regard, it contrasts sharply with numerous other efforts to solve adaptive systems problem by merely "saving and mutating the best", such as the 1966 *Artificial Intelligence through Simulated Evolution* (Fogel et. al.) and other work using only asexual mutation (Dawkins 1987, Lenat 1976, Lenat 1983, Lenat 1984, Hicklin 1986).

Recent work on genetic algorithms is surveyed in the recent special issue of *Machine Learning* journal on genetic algorithms (Goldberg and Holland 1988) as well as in Goldberg (1989), Schaffer (1989), Grefenstette (1987a), Grefenstette (1985), Davis (1987), and the survey article by Booker et. al. in the recent special issue on machine learning of *Artificial Intelligence* journal (1989).

Representation is a key issue in genetic algorithm work because the representation scheme can severely limit the window by which the system observes its world. However, as Davis and Steenstrup (1987) point out, "In all of Holland's work, and in the work of many of his students, chromosomes are bit strings." String-based representation schemes are difficult and unnatural for many problems and the need for more powerful representations has been recognized for some time (De Jong 1985, De Jong 1987, De Jong 1988).

Smith (1980 and 1983) departed from the early fixed-length character strings by introducing variable length strings, including strings whose elements were if-then rules (rather than single characters). Various application-specific variations on the basic genetic algorithm are reported in

Grefenstette (1985) and Grefenstette (1987a). These variations include both work where the structures undergoing adaptation are more complex than strings and work where the genetic operations are modified to meet the needs of particular applications.

The introduction of the classifier system (Holland 1986, Holland et. al. 1986, Holland and Burks 1987, Holland and Burks 1989) continued the trend towards increasing the complexity of the structures undergoing adaptation. A classifier system is a cognitive architecture into which the genetic algorithm has been embedded so as to allow adaptive modification of a population of string-based if-then rules (whose condition and action parts are fixed length binary strings). The classifier system architecture blends the desirable features of if-then rules from expert systems, a more precisely targeted allocation of credit to specific rules for performance, and the creative power of the genetic algorithm. In addition, embedding the genetic algorithm into the classifier system architecture creates a computationally complete system which can, for example, realize functions such as the exclusive-or function. The exclusive-or function was not realizable by early single layer linear perceptrons (Minsky and Papert 1969) and, because the exclusive-or function yields totally uninformative schemata (similarity templates), it was not realizable with conventional linear genetic algorithms using fixed length binary strings.

Friedberg's early efforts (1958, 1959) at generating computer programs in a simplified assembly language did not use the genetic algorithm; however, Cramer (1985) used the genetic algorithm to generate computer programs with a fixed structure and reported on the highly epistatic nature of the problem of generating programs. Hicklin (1986) applied a mutation operation to LISP programs. Fujiki (1986) discussed the desirability of applying all the genetic operations to LISP programs. Subsequently, Fujiki and Dickinson (1987) implemented genetic operations to manipulate the if-then clauses of a LISP computer program consisting of a single COND (conditional) statement specifying strategies for playing the iterated prisoner’s dilemma game.

String-based representation schemes do not provide the hierarchical structure central to the organization of computer programs (into programs and subroutines) and the organization of behavior (into tasks and subtasks). Some work has been done to alleviate this problem, notably Wilson's modification (1987b) of Holland’s bucket brigade algorithm for credit allocation in classifier systems. Wilson’s hierarchical credit allocation algorithm was designed to encourage the creation of hierarchies of rules and discourage the exceedingly long linear sequences of rules that are otherwise characteristic of classifier systems.

String-based representation schemes do not provide any convenient way of representing arbitrary computational procedures or of incorporating iteration or recursion when these capabilities are inherently necessary to solve the problem.

Moreover, string-based representation schemes do not have dynamic variability so that the initial selection of string length limits in advance the number of internal states of the system and the computational complexity of what the system can learn.

Moreover, string-based representation schemes do not facilitate representation of situations where computer programs modify and execute themselves. The predetermination of the size and shape of solutions and the pre-identification of the particular components of solutions has been a bane of machine learning systems from the earliest times (Samuel 1959) as well as in later efforts in automatic programming (Green et. al. 1974, Lenat 1976, Lenat and Brown 1984).

3. THE "GENETIC PROGRAMMING" PARADIGM

In this section we describe the "genetic programming" paradigm using hierarchical genetic algorithms by specifying (1) the nature of the structures that undergo adaptation in this paradigm, (2) the search space of structures, (3) the initial structures, (4) the environment and fitness function which evaluates the structures in their interaction with the environment, (5) the operations that are performed to modify the structures, (6) the state (memory) of the algorithmic system at each point in time, (7) the method for terminating the algorithm and identifying its output, and (8) the parameters that control the process.

3.1. THE STRUCTURES UNDERGOING ADAPTATION

The structures that undergo adaptation in the genetic programming paradigm are hierarchically structured computer programs whose size, shape, and complexity can dynamically change during the process. This is in contrast to the one-dimensional linear strings (whether of fixed or variable length) of characters (or other objects) cited previously.

The set of possible structures that undergo adaptation in the genetic programming paradigm is the set of all possible composition of functions that can be composed recursively from the available set of n functions $F = \{f_1, f_2, ... , f_n\}$ and the available set of m terminals $T = \{a_1, a_2, ... , a_m\}$. Each particular function f in F takes a specified number z(f) of arguments $b_1, b_2, ..., b_{z(f)}$. Depending on the particular problem of interest, the functions may be standard arithmetic operations (such as addition, subtraction, multiplication, and division), standard mathematical functions(such as SIN, EXP, etc.), Boolean operations, domain-specific functions, logical operators such as If-Then-Else, and iterative operators such as Do-Until, etc. We assume that each function in the function set of well defined for any value in the range of any of the functions. The "terminals" may be variable atomic arguments, such as the state variables of a system; constant atomic arguments, such as 0 and 1; and, in some cases, may be other atomic entities such as functions with no arguments (either because the argument is implicit or because the real functionality of the function is the side effect of the function on the state of the system).

Virtually any programming language is capable of expressing and evaluating the compositions of functions described above (e.g. PASCAL, FORTRAN, C, FORTH, LISP, etc.). We have chosen the LISP programming language (first developed by John McCarthy in the 1950's) for the work described in this article for the following six reasons.

First, both programs and data have the same form in LISP. This means that it is possible to genetically manipulate a computer program and then immediately execute it (using the EVAL function of LISP).

Second, the above-mentioned common form for both programs and data in LISP is equivalent to the parse tree for the computer program. In spite of their outwardly different appearance and syntax, most "compiled" programming languages convert, at the time of compilation, a given program into a parse tree representing its underlying composition of functions. In most programming languages, this parse tree is not accessible to the programmer. As will be seen, we need access to the parse tree because we want to genetically manipulate the sub-parts of given computer programs (i.e. sub-trees of the parse tree). LISP gives us convenient access to this parse tree.

Third, LISP facilitates the programming of structures whose size and shape changes dynamically (rather than predetermined in advance). Moreover, LISP's dynamic storage allocation and garbage collection provides administrative support for programming of dynamically changing structures.

Fourth, LISP facilitates the handling of hierarchical structures.

Fifth, the LISP programming language is reentrant.

Sixth, software environments with a rich collection of tools are commercially available for the

LISP programming language.

For these reasons, we have chosen the LISP programming language for the work described in this paper. In particular, we have chosen the Common LISP dialect of LISP (Steele 1984). That is, the structures that undergo adaptation in the genetic programming paradigm are LISP computer programs (i.e. LISP symbolic expressions).

It is important to note that we did not choose the LISP programming language for the work described in this article because we intended to make any use of the list data structure or the list manipulation functions unique or peculiar to the LISP programming language.

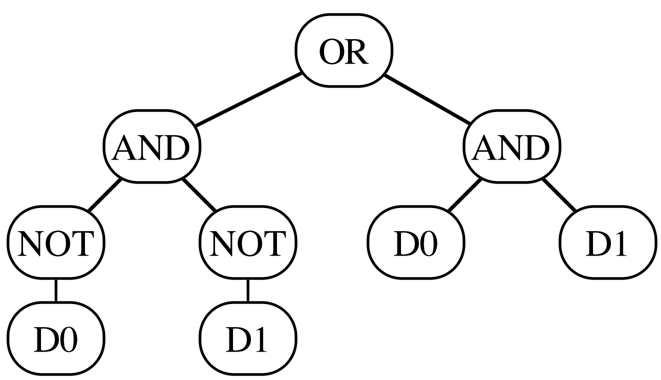
The general nature of the LISP programming language can be illustrated by a simple example. For example, (+ 1 2) is a LISP symbolic expression (S-expression) that evaluates to 3. In this S-expression, the addition function (+) appears just inside the left-most parenthesis of the S-expression. This "prefix" form (e.g. Polish notation) represents the application of a function (+) to its arguments (1 and 2) and is a convenient way to express a composition of functions. Thus, the S-expression (+ 1 (* 2 3)) is a composition of two functions (+ and *) that evaluates to 7. Similarly, the S-expression (+ 1 2 (IF (> TIME 10) 3 4)) demonstrates the “function” > being applied to the variable atom TIME and the constant atom 10. The sub-expression (> TIME 10) evaluates to either T (True) or NIL (False) and this value becomes the first argument of the “function” IF. The function IF returns either its second argument (i.e. the constant atom 3) if its first argument is T and it returns its third argument (i.e. the constant atom 4) if its first argument is NIL. Thus, this S-expression evaluates to either 6 or 7 depending on the current value of TIME.

Now consider the Boolean exclusive-or function which can be expressed in disjunctive normal form and represented as the following LISP S-expression:

```
(OR (AND (NOT D0) (NOT D1)) (AND D0 D1)).
```

The set of functions here is F = {AND, OR, NOT} and the set of terminals is T = {D0, D1}. For our purposes here, terminals can be viewed as functions requiring zero arguments in order to be evaluated. Thus, we can combine the set of functions and terminals into a combined set $C = F \cup T =$ {AND, OR, NOT, D0, D1} taking 2, 2, 1, 0, and 0 arguments, respectively.

Any LISP S-expression can be graphically depicted as a rooted point-labeled tree with ordered branches. The tree corresponding to the LISP S-expression above for the exclusive-or function is shown below:



In this graphical depiction, the 5 internal points of the tree are labeled with functions (e.g. OR, AND, NOT, NOT, and AND); the 4 external points (leaves) of the tree are labeled with terminals (e.g. the variable atoms D0, D1, D0, and D1); and the root of the tree is labeled with the function (i.e. OR) appearing just inside the outermost left parenthesis of the LISP S-expression. This tree is equivalent to the parse tree which most compilers construct internally to represent a given computer program.

Note that the set of functions and terminals being used in a particular problem should be selected so as to be capable of solving the problem (i.e. some composition of the available functions and terminals should yield a solution). Removing the function NOT from the function set F above would, for example, create an insufficient function set for expressing the Boolean exclusive-or function.

3.2. THE SEARCH SPACE

The search space for the genetic programming paradigm is the hyperspace of valid LISP S-expressions that can be recursively created by compositions of the available functions and available terminals for the problem. This search space can, equivalently, be viewed as the hyperspace of rooted point-labeled trees with ordered branches having internal points labeled with the available functions and external points (leaves) labeled with the available terminals.

3.3. THE INITIAL STRUCTURES

Generation of the initial random population begins by selecting one of the functions from the set F at random to be the root of the tree. Whenever a point is labeled with a function (that takes k arguments), then k lines are created to radiate out from the point. Then, for each line so created, an element is selected at random from the entire combined set C to be the label for the endpoint of that line. If an terminal is chosen to be the label for any point, the process is then complete for that portion of the tree. If a function is chosen to be the label for any such point, the process continues. The probability distribution over the terminals and functions in the combined set C and the number of arguments taken by each function implicitly determines an average size for the trees generated by this initial random generation process. In this paper, this distribution is always a uniform random probability distribution over the entire set C (with the exception that the root of the tree must be a function). In some problems, one might bias this initial random generation process with a non-uniform distribution or by seeding particular individuals into the population.

3.4. THE FITNESS FUNCTION

Each individual in a population is assigned a fitness value as a result of its interaction with the environment. Fitness is the driving force of Darwinian natural selection and, likewise, of genetic algorithms.

The environment is a set of cases which provides a basis for evaluating the fitness of the S-expressions in the population. For example, for the exclusive-or function, the obvious choice for the environment is the set of four combinations of possible values for the two variable atoms D0 and D1 along with the associated value of the exclusive-or function for the four such combinations.

For most of the problems described herein, the “raw fitness” of any LISP S-expression is the sum of the distances (taken over all the environmental cases) between the point in the range space returned by the S-expression for a given set of arguments and the correct point in the range space. The S-expression may Boolean-valued, integer-valued, real-valued, complex-valued, vector-valued, multiple-valued, or symbolic-valued.

If the S-expression is integer-valued or real-valued, the sum of distances is the sum of absolute values of the differences between the numbers involved. In particular, the raw fitness r(i,t) of an individual LISP S-expression i in the population of size M at any generational time step t is

$$r(i,t) = \sum_{j=1}^{N_e} |S(i,j) - C(j)|$$

where S(i,j) is the value returned by S-expression i for environmental case j (of N_e environmental cases) and where C(j) is the correct value for environmental case j.

If the S-expression is Boolean-valued or symbolic-valued, the sum of distances is equivalent to the number of mismatches. If the S-expression is complex-valued, or vector-valued, or multiple-valued, the sum of the distances is the sum of the distances separately obtained from each component of the vector or list.

The closer this sum of distances is to zero, the better the S-expression.

If the S-expression (or each component of a vector or list) is real-valued or integer-valued, the square root of the sum of squares of distances can, alternatively, be used to measure fitness (thereby increasing the influence of more distant points).

For some problems described herein, the fitness function does not work with the actual value returned by the individual S-expression, but some number (e.g. elapsed time, total score, cases handled, etc.) which is indirectly created by the action of the S-expression. Nonetheless, the raw fitness function must be defined in such a way that the raw fitness is to be smaller for better individuals in the population.

For each problem described herein, the raw fitness r(i,t) is then adjusted (scaled) to produce an adjusted fitness measure a(i,t). The “adjusted fitness” value a(i,t) is

$$a(i,t) = \frac{1}{(1+r(i,t))},$$

where r(i,t) is the raw fitness for individual i at time t. Unlike raw fitness, the adjusted fitness is larger for better individuals in the population. Moreover, the adjusted fitness lies between 0 and 1.

Each such adjusted fitness value a(i,t) is then normalized. The “normalized fitness” value n(i,t) is

$$n(i,t) = \frac{a(i,t)}{\sum_{k=1}^{M} a(k,t)}$$

The normalized fitness not only ranges between 0 and 1 and is larger for better individuals in the population, but the sum of the normalized fitness values is 1. When we use the phrases "proportional to fitness" or "fitness proportionate" in this paper, we are referring to normalized fitness.

As will be seen, it is also possible for the fitness function to consider secondary factors (e.g., efficiency of the S-expression, parsimony of the S-expression, compliance with the initial conditions of a differential equation, etc.).

When the number of environmental cases is large, it is sometimes expeditious to compute the fitness function using only a sampling of the possible environmental cases (including, possibly, a sampling that varies from generation to generation to minimize the possible bias resulting from such sampling).

3.5. THE OPERATIONS THAT MODIFY THE STRUCTURES

The two primary operations for modifying the structures undergoing adaptation are Darwinian fitness proportionate reproduction and crossover (recombination). They are described below. Additional operations are described in section 7.

3.5.1. THE FITNESS PROPORTIONATE REPRODUCTION OPERATION

The operation of fitness proportionate reproduction for the genetic programming paradigm is the basic engine of Darwinian reproduction and survival of the fittest. It operates on only one parental S-expression and produces only one offspring S-expression each time it is performed. That is, it is an asexual operation. If $f(s_i(t))$ is the fitness of individual s_i in the population at generation t, then, each time this operation is performed, each individual in the population has a probability of

$$\frac{f(s_i(t))}{\sum_{j=1}^{M} f(s_j(t))}$$

of being copied into the next generation by the operation of fitness proportionate reproduction.

Note that the parents remain in the population while this operation is performed and therefore can potentially participate repeatedly in this operation (and other operations) during the current generation. That is, the selection of parents is done with replacement (i.e. reselection) allowed.

3.5.2. THE CROSSOVER (RECOMBINATION) OPERATION

The crossover (recombination) operation for the genetic programming paradigm creates variation in the population by producing offspring that combine traits from two parents. The crossover operation starts with two parental S-expressions and produces at least one offspring S-expression. That is, it is a sexual operation. In this paper, two offspring will be produced on each occasion that the crossover operation is performed.

In general, at least one parent is chosen from the population with a probability equal to their respective normalized fitness values. In this paper, both parents are so chosen.

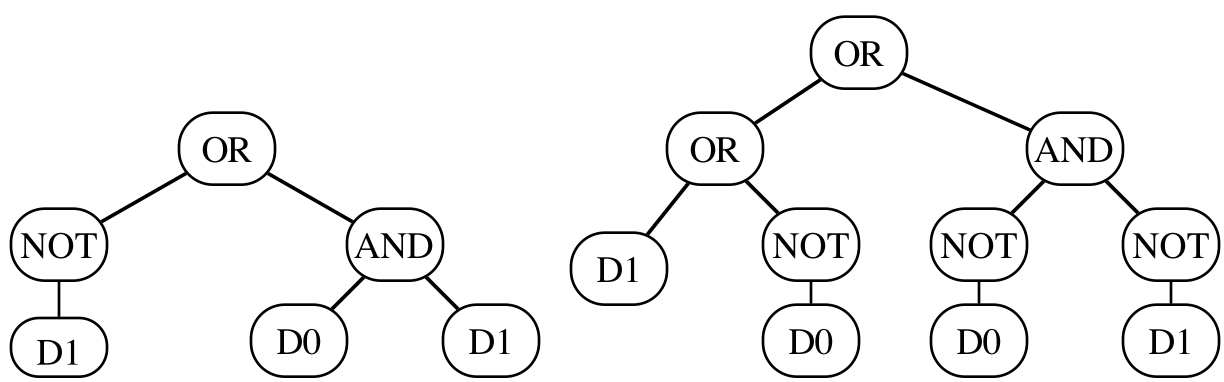
The operation begins by randomly and independently selecting one point in each parent using a probability distribution. Note that the number of points in the two parents typically are not equal.

As will be seen, the crossover operation is well-defined for any two S-expressions. That is, for any two S-expressions and any two crossover points, the resulting offspring are always valid LISP S-expressions. Offspring consist of parts taken from each parent.

The "crossover fragment" for a particular parent is the rooted sub-tree whose root is the crossover point for that parent and where the sub-tree consists of the entire sub-tree lying below the crossover point (i.e. more distant from the root of the original tree). Viewed in terms of lists in LISP, the crossover fragment is the sub-list starting at the crossover point.

The first offspring is produced by deleting the crossover fragment of the first parent from the first parent and then impregnating the crossover fragment of the second parent at the crossover point of the first parent. In producing this first offspring the first parent acts as the base parent (the female parent) and the second parent acts as the impregnating parent (the male parent). The second offspring is produced in a symmetric manner.

For example, consider the two parental LISP S-expressions below.



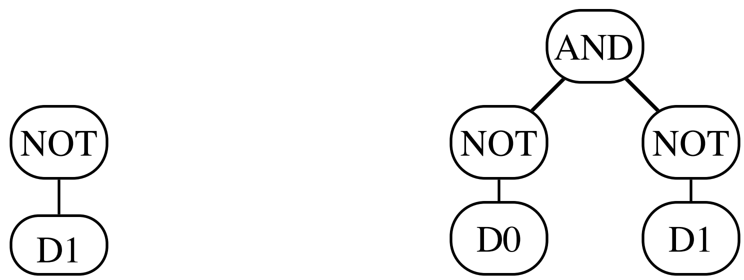
In terms of LISP S-expressions, the two parents are

```
(OR (NOT D1) (AND D0 D1))
```

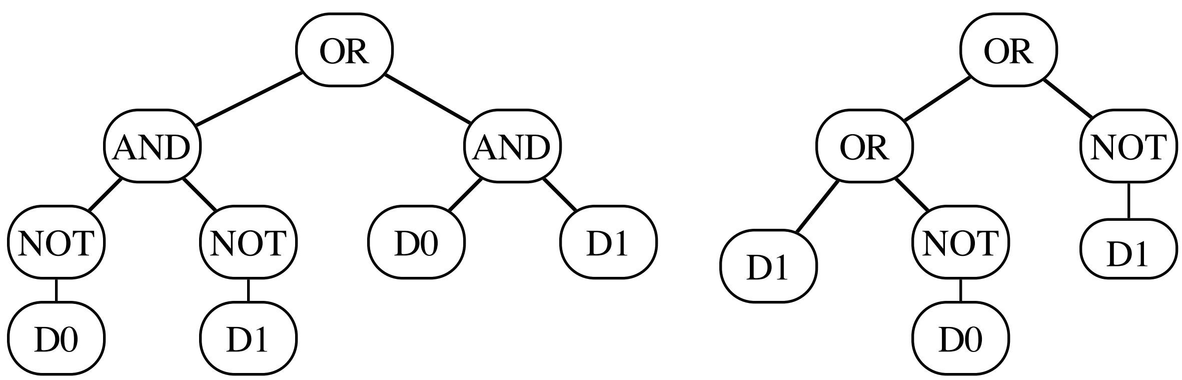
and

`(OR (OR D1 (NOT D0))` **`(AND (NOT D0) (NOT D1))`**`)`

Assume that the points of both trees above are numbered in a depth-first way starting at the left. Suppose that the second point (out of the 6 points of the first parent) is selected as the crossover point for the first parent and that the sixth point (out of the 10 points of the second parent) is selected as the crossover point of the second parent. The crossover points are therefore the NOT function in the first parent and the AND function in the second parent. Thus, the bold, underlined portion of each parent above are the crossover fragments. The two crossover fragments are shown below:



The two offspring resulting from crossover are shown below.



Note that the first offspring above is a perfect solution for the exclusive-or function, namely

`(OR (AND (NOT D0) (NOT D1)) (AND D0 D1)).`

Note that because entire sub-trees are swapped, this genetic crossover (recombination) operation produces valid LISP S-expressions as offspring regardless of which point is selected in either parent.

If the root of one tree happens to be selected as the crossover point, the crossover operation will insert that entire parent into the second tree at the crossover point of the second parent. In addition, the sub-tree from the second parent will, in this case, then become the second offspring.

If the roots of two parents happen to be chosen as crossover points, the crossover operation simply degenerates to an instance of fitness proportionate reproduction on those two parents.

Note that if an individual mates with itself, the two resulting offspring will generally be different (if the crossover points selected are different).

If a terminal is located at the crossover point in precisely one parent, then the sub-tree from the second parent is inserted at the location of the terminal in the first parent and the terminal from the first parent is inserted at the location of the sub-tree in the second parent. In this case, the crossover operation often has the effect of increasing the depth of one tree and decreasing the depth of the second tree.

If terminals are located at both crossover points selected, the crossover operation merely swaps these terminals from tree to tree.

3.6. THE STATE OF THE SYSTEM

The state of the hierarchical genetic algorithm system at any generation consists only of the current population of individuals in the population. There is no additional memory or centralized bookkeeping used in directing the adaptive process.

3.7. IDENTIFYING THE RESULTS AND TERMINATING THE ALGORITHM

The solution produced by this algorithm at any given time can be viewed as the entire population of disjunctive alternatives (presumably with improved overall average fitness) or, more commonly, as the single best individual in the population at that time ("winner takes all"). The algorithm can be terminated when either a specified total number of generations have been run or when some performance criterion is satisfied. In many problems, this performance requirement for termination may be that the sum of the distances reaches a value of zero. If a solution can be recognized when it is encountered, the algorithm can be terminated at that time and the single best individual can be considered as the output of the algorithm.

3.8. THE PARAMETERS THAT CONTROL THE ALGORITHM

The algorithm is controlled by various parameters, including two major parameters and five minor parameters.

The two major parameters are the population size and the number of generations to be run.

A population size of 300 was used for all problems described in section 4 with the exception of the 11-multiplexer problem. After the Boolean 6-multiplexer was solved using the common population size of 300, we noted that the search space of the next larger version of multiplexer problem (i.e. a search space of size approximately 10^{616} for the 11-multiplexer problem) would alone indicate using a larger population size for this particular problem. An especially large population size (i.e. 4000) was then chosen for this particular problem in order to force down the number of generations required to arrive at a solution so that it would be practical to create a complete genealogical audit trail for this problem.

The number of generations was 51 (i.e. an initial random generation and 50 subsequent generations). Note if termination of the algorithm is under control of some performance criterion (which was not the case in this paper), this parameter merely provides an overall maximum number of generations to be run.

These two major parameters are the same for all problems described in section 4 (with the one exception already noted).

The five minor parameters are discussed in section 6.3 and are the same for all problems described in this article.

There is no reason to think that the selection of the two major parameters and five minor parameters used in this paper are optimal. In fact, it is clear that computational resources were needlessly wasted on some of the problems described in this paper and convergence was needlessly hampered on some of the problems by both the specific choice of parameters and by the decision to use the same parameters on all of the problems described in this article. The problem of optimal parameter selection is an interesting and important issue affecting both string-based genetic algorithms and hierarchical genetic algorithms (as well as other machine learning paradigms). The problem of optimal parameter selection is deserving of future work. The discovery of a theoretical framework for making parameter selections in a optimal way would be most welcome. However, the purpose of this paper is not to study the optimization of parameters, but to illustrate that a surprising range of seemingly different problems can be couched and solved using the new, common, unified approach described herein. This common approach consists of first viewing the original problem as a problem requiring discovery of a computer program to solve the problem (thus making the original problem into a problem of searching a space of possible computer programs for a most fit individual computer program). Then, the genetic programming paradigm provides a way of finding that individual computer program. The fact that the <u>same</u> (admittedly non-optimal and essentially arbitrary) set of parameters were used on <u>all</u> problems described in section 4 of this paper should help focus attention on the fact that a variety of different problems from different areas were successfully handled and that this result did not depend on parameter selection.

4. EXPERIMENTAL RESULTS

This section describes some experiments in machine learning using the "genetic programming" paradigm. In this section of this paper, we show (1) how we approached each problem so that it could be solved using the "genetic programming" paradigm and (2) the result of one particular run for each problem.

The experiments have been selected to illustrate a variety of different types of problems from various different areas. The sample problems selected involve functions that are integer-valued, real-valued, Boolean-valued, vector-valued, complex-valued, and symbolic-valued. Some of the problems require recursion for their solution. Some of the problems require one of three types of iteration described in this paper. Some of the problems involve functions whose real functionality is the side effects they cause on the state of the system involved, rather than their return values. Some problems involve the use of functional abstractions that give a name to a useful sub-program or computed value so it can be subsequently used elsewhere in the computer program.

In order to provide a degree of comparability with previous research, many of the problems described in this paper benchmark problems that have been previously reported in the literature in connection with existing paradigms for machine learning, artificial intelligence, induction, neural nets, decision trees, and classifier systems. In many cases, these previously studied problems present certain difficulties for paradigms for machine learning, artificial intelligence, induction, neural nets, and classifier systems.

Since the algorithm is probabilistic, we almost never get the precisely same result twice. Moreover, we almost never get the solution to the problem in the form we contemplated (although these solutions may be equivalent to what we contemplated). Thus, no one particular run and no particular solution is truly typical or representative of the others. The purpose of this section is to illustrate the genetic programming paradigm by showing one particular illustrative result from one particular run. In choosing the particular result and run for the illustrative purposes of this section, we have avoided showing the "prettiest" solution and we have similarly avoided showing the most convoluted solution. In section 9, we present one way of statistically measuring the performance of the "genetic programming" paradigm, namely, the amount of computer processing required to produce a solution with 99% probability (given that each run involved a particular population size and given that each run was cut off after a particular number of generations).

Each of the problems presented has been repeatedly solved on numerous occasions (typically dozens or hundreds of times). For each experiment reported below, the author believes that sufficient information is provided to allow the experiment to be independently replicated to produce substantially similar results (within the limits inherent in any process involving stochastic operations).

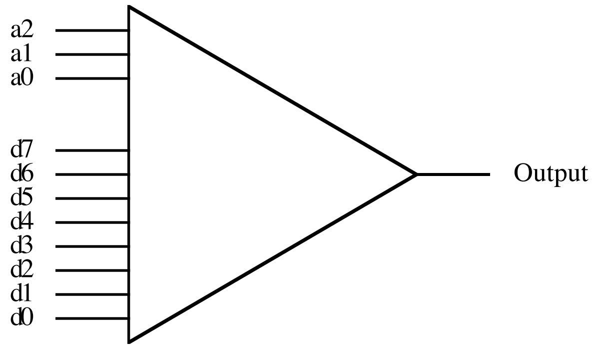
A discussion of computer implementation, the computer interface used to monitor runs, and the unique programming problems associated with executing ill-behaved, randomly-generated, genetically-modified computer programs is found in Section 6.

4.1. MACHINE LEARNING OF A FUNCTION

The problem of machine learning of a function requires developing a composition of functions that can return the correct value of the function after seeing only a relatively small number of specific examples of the value of the function associated with particular combinations of arguments.

4.1.1. BOOLEAN 11-MULTIPLEXER FUNCTION

In this first experiment, the problem is to learn the Boolean 11-multiplexer function. In general, the input to the Boolean multiplexer function consists of k "address" bits a_i and 2^k "data" bits d_i and is a string of length $k+2^k$ of the form $a_{k-1}...a_1a_0\, d_{2^k-1}...d_1\, d_0$. The value of the multiplexer function is the value (0 or 1) of the particular data bit that is singled out by the k address bits of the multiplexer. For example, for the 11-multiplexer (where k = 3), if the three address bits $a_2a_1a_0$ are 110, then the multiplexer singles out the sixth data bit d_6 to be its output.



We present this first example in considerable detail in order to show the interplay in the genetic programming paradigm of the genetic variation inevitably created in the initial random generation, the small improvements via localized hill-climbing from generation to generation, the way particular individuals become specialized and able to correctly handle certain sub-cases of the problem (case-splitting), the creative role of crossover in recombining valuable parts of more fit parents, and how the nurturing of a large population of disjunctive alternative solutions to the problem (rather than a single point in the solution space) helps avoid false peaks.

We present the Boolean multiplexer problem here first for several reasons. First, the problem has an easily quantifiable search space. Secondly, it is especially easy to understand how the components of a Boolean expression contribute to the performance of the overall Boolean expression (in contrast to many of the other problems discussed herein). Third, Boolean problems present fewer computer implementation problems than the other problems discussed later in this paper. Fourth, the Boolean multiplexer problem has been previously used as a test function in studies of neural nets (Barto et. al. 1985), classifier systems (Wilson 1987a), and decision trees (Quinlan 1988).

The first step in applying the "genetic programming" paradigm to a problem is to select the set of functions and arguments that will be available to the algorithm for constructing the computer programs (S-expressions) that will try to solve the problem. For some problems (in particular, Boolean function learning problems), this choice is especially straight-forward and obvious.

The set of available functions for this problem is F = {AND, OR, NOT, IF}. This set of basic logical functions is sufficient for any problem involving a Boolean function, and, in addition, is a convenient set in that it often produces easily understood S-expressions. The AND and OR functions take two arguments. The NOT function takes one argument. The IF function is the Common LISP IF-THEN-ELSE function and takes three arguments. We use this function set for all Boolean problems described in this paper.

The set of available terminals for this problem has 11 elements which correspond to the 11 inputs to the Boolean 11-multiplexer. That is, the terminal set T = {A0, A1, A2, D0, D1, ... , D7}.

The potential set of structures undergoing adaptation in this problem is the set of all LISP S-expressions that can be recursively composed from the set of available functions and the set of available terminals.

The Boolean multiplexer function with $k+2^k$ arguments is one of 2^{k+2^k} possible Boolean functions of $k+2^k$ arguments. Thus, the search space for the Boolean multiplexer is of size 2^{k+2^k}. When k=3, this search space is of size $2^{2^{11}}$, i.e. 2^{2048} , which is approximately 10^{616}. Every possible Boolean function of $k+2^k$ arguments can be realized by at least one LISP S-expression composed from the functions and terminals above. Disjunctive normal form is one such way.

The environment consists of the 2^{11} possible combinations of the 11 arguments $a_0a_1a_2d_0d_1d_2d_3d_4d_5d_6d_7$ along with the associated correct value of the 11-multiplexer function. For the 11-multiplexer (where k = 3), there are 2048 such combinations of arguments in the environment. For this particular problem, we use the entire set of 2048 combinations of arguments (i.e. we do not use sampling of the environment).

The raw fitness of a LISP S-expression is the sum of the distances (taken over all the environmental cases) between the point returned by the S-expression for a given set of arguments and the correct point. When a Boolean value is returned by an S-expression, this distance is 1 if the return value matches the correct value and is 0 if it does not match. Thus, the sum of the distances over all the environmental cases is equivalent to the number of mismatches. Thus, the raw fitness of an S-expression can range over 2049 different values between 0 and 2048. A raw fitness of 0 denotes a 100% correct individual. Note that whenever the fitness in a particular problem takes on a specific value (e.g. 0) for a 100% correct individual, we have a way to recognize attainment of a solution to the problem.

We have found it highly useful to define an auxiliary measure for monitoring the progress of runs which counts the number of “hits” between an S-expression and the correct value. For this problem, the number of “hits” is simply 2048 minus the raw fitness (mismatches). For problems involving integer values discussed later, this auxiliary measure counts the number of “hits” between an S-expression and the correct environmental value (whereas the raw fitness is a cumulative distance measure). For real-valued problems discussed later, this auxiliary measure counts the number of “near hits” for which the S-expression comes within a small specified tolerance (called the "criterion") of the correct environmental value. These “hits” or “near hits” measures are used only for monitoring runs and are not accessible to, or used by, the genetic algorithm for any problem discussed herein.

We illustrate the overall process by discussing one particular run of the Boolean 11-multiplexer in detail. The process begins with the generation of the initial random population (i.e. generation 0).

Predictably, the initial random population includes a variety of highly unfit individuals. Some individual S-expressions involve logical contradictions such as `(AND A0 (NOT A0))`. Others involve inefficiencies such as `(OR D7 D7)`. Some are passive and merely pass an input through as the output, such as `(NOT (NOT A1))`. Some of the initial random individuals base their decision on precisely the wrong arguments (i.e. data bits), such as `(IF D0 A0 A2)`. Most of the initial random individuals are partially "blind" in that they do not incorporate all 11 arguments that are known to be necessary for a solution to the problem. Some S-expressions are just nonsense, such as `(IF (IF (IF D2 D2 D2) D2) D2)`.

Nonetheless, even in this highly unfit initial random population, some individuals are somewhat more fit than others. For the run in question, the individuals in the initial random population (generation 0) had raw fitness values ranging from 768 mismatches (i.e. 1280 hits) to 1280 mismatches (i.e. 768 hits). As it happens, a total of 25 individuals out of 4000 tied with the high score of 1280 hits on generation 0. One such individual was the S-expression `(IF A0 D1 D2)`. In spite of its obvious shortcomings (e.g. it is partially blind in that it uses only 3 of the 11 necessary terminals of the problem), this individual nonetheless does some things right. It uses an address bit (A0) as the basis for its action. In addition, it uses data bits (D1 and D2) as its output. Moreover, if A0 (which is the low order binary bit of the 3-bit address) is T (True), an odd numbered data bit (D1) is selected, while if A0 is NIL, an even numbered data bit (D2) is selected. This individual is far from perfect, but, it is more fit than any of the others.

The worst individual in the population was `(OR (NOT A1) (NOT (IF (AND A2 A0) D7 D3)))` and had a raw fitness of 1280 (i.e. only 768 hits). The average raw fitness for generation 0 is 985.4.

The “hits” histogram provides additional details about the population for a particular generation.

The horizontal axis is the number of hits and the vertical axis is the number of individuals in the population scoring that number of hits. For example, a total of 50 different levels of raw fitness are represented in the population at generation 0. 1490 individuals had raw fitness 1152 (the highest score for generation 0). The figure below shows the histogram of the population for generation 0 of this run of this problem.

A new population is then created from the current population using the operations of fitness proportionate reproduction and crossover. When these operations are completed, the new population (i.e. the new generation) replaces the old population.

Starting with generation 1, the average raw fitness of the population immediately begins improving (i.e. decreasing) from the baseline value for generation 0 of 985.4 to about 891. We typically see this kind of generally improving trend in average raw fitness from generation to generation. As it happens, in this particular run, the average raw fitness improves monotonically between generation 2 and generation 9 and assumes values of 845, 823, 763, 731, 651, 558, 459, and 382.

At the same time, we typically see a generally improving trend in the raw fitness of the best individual in the population from generation to generation. As it happens, in this particular run, the raw fitness (i.e. number of mismatches) of the best single individual in the population improves monotonically between generation 2 and generation 9 and assumes values of 640 (i.e. 1408 hits), 576, 384, 384, 256, 256, 128, and 0 (i.e. a perfect score of 2048 hits), respectively. On the other hand, the raw fitness of the worst individual in the population typically fluctuates considerably. For this particular run, this number starts at 1280 and actually deteriorates to 1792 (only 256 hits out of 2048) by generation 9.

The number of hits for the best single individual in the population rises to 1408 for generations 1 and 2 of the run. In generation 1, for example, one individual in the population had a score of 1408, namely `(IF A0 (IF A2 D7 D3) D0)`. Note that this individual performs better than the best individual from generation 0 because it considers two address bits (A0 and A2) in deciding which data bit to choose as output and because it incorporates three data bits as its potential output. In contrast, the best individual in generation 0 considered only one address bit (A0) and incorporated only two data bits as potential output. Although still far from perfect, the best individual from generation 1 is less blind and more complex than the best individual of the previous generation.

By generation 2, the number of individuals sharing this high score of 1408 hits rose to 21. The high point of the histogram for generation 2 has advanced from 1152 for generation 0 to 1280. There are now 1620 individuals with 1280 hits.

In generation 3, one individual in the population attained a new high score of 1472 hits. This individual is

```
(IF A2 (IF A0 D7 D4)
       (AND (IF (IF A2 (NOT D5) A0) D3 D2) D2)).
```

Generation 3 shows further advances in fitness for the population as a whole. The number of individuals with a fitness of 1280 (the high point for generation 2) has risen to 2158 for generation 3. Moreover, the center of gravity of the fitness histogram has shifted from left to right. In particular, the number of individuals with fitness 1280 or better has risen from 1679 in generation 2 to 2719 in generation 3.

In generations 4 and 5, the best individual has a score of 1664 hits. This score is attained by one individual in generation 4 and by 13 individuals in generation 5. This best individual is

```
(IF A0 (IF A2 D7 D3) (IF A2 D4 (IF A1 D2 (IF A2 D7 D0)))).
```

Note that this individual uses all three address bits (A2, A1, and A0) in deciding upon the output. It also uses five of the eight data bits. By generation 4, the high point of the histogram has moved to 1408 with 1559 individuals.

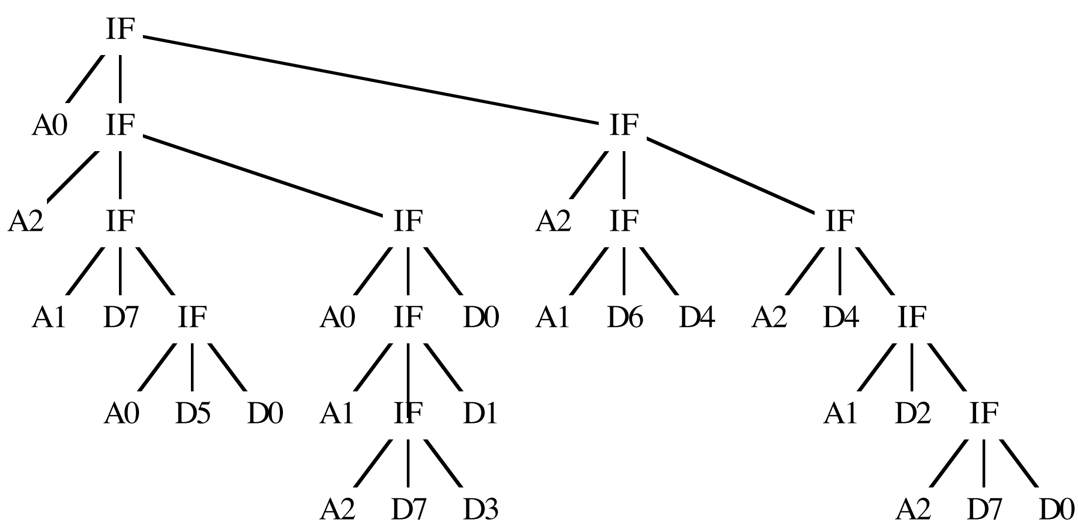
In generation 6, four individuals attain a score of 1792. The high point of the histogram has moved to 1536. In generation 7, 70 individuals attain this score of 1792.

In generation 8, four individuals attain a score of 1920. The high point of the histogram has moved to 1664 and 1672 individuals share this value. Moreover, an addition 887 individuals score 1792.

In generation 9, one individual emerges with a l00% perfect score of 2048 hits. That individual is

```
(IF A0 (IF A2 (IF A1 D7 (IF A0 D5 D0))
              (IF A0 (IF A1 (IF A2 D7 D3) D1) D0))
       (IF A2 (IF A1 D6 D4)
              (IF A2 D4 (IF A1 D2 (IF A2 D7 D0)))))
```

This 100% correct individual is depicted graphically below:



Thus, this 100% correct individual can be simplified (either manually or via the editing operation described later) to

```
(IF A0 (IF A2 (IF A1 D7 D5) (IF A1 D3 D1))
       (IF A2 (IF A1 D6 D4) (IF A1 D2 D0))).
```

When so rewritten, it can be seen that this individual correctly performs the 11-multiplexer function by first examining address bits A0, A2, and A1 and then choosing the appropriate one of the eight possible data bits.

As one progresses from generation 0 through 9, there is a left-to-right “slinky” movement of the “center of mass” of the histogram, the single best individual, and the high point of the histogram.

Further insight can be gained by studying the genealogical audit trail of the process. This audit trail consists of a complete record of the details of each instance of the operations. In the case of the operations of fitness proportionate reproduction, and crossover, the details consist of the individual(s) chosen for the operation and the particular point chosen within each such participating individual(s).

Construction of the audit trail starts with the individuals of the initial random generation (generation 0). Certain additional information such as the individual's rank location in the population (after sorting by normalized fitness) and its raw fitness is also carried along as a convenience in interpreting the genealogy. Then, as each operation is performed to create a new individual for the next generation, a list is recursively formed consisting of the operation performed, the individual(s) participating in the operation, the details of that operation, and, finally, a pointer to the audit trail(s) accumulated so far for the individual(s) participating in that operation.

An individual occurring at generation h has up to 2^{h+1} ancestors. The number of ancestors is less than 2^{h+1} to the extent that operations other than crossover are involved; however, crossover is, by far, the most frequent operation. For example, an individual occurring at generation 9 has up to 1024 ancestors. Note that a particular ancestor often appears more than once in this genealogy because all selections of individuals to participate in the basic genetic operations are skewed in proportion to fitness. Moreover, even for a modest sized value of h, 2^{h+1} will typically be greater than the population size. This repetition, of course, does nothing to reduce the size of the genealogical tree.

Even with the use of pointers from descendants back to ancestors, construction of a complete genealogical audit trail is exponentially expensive in both computer time and memory space. Note that the audit trail must be constructed for each individual of each generation because the identity of the l00% correct individual(s) eventually solving the problem is not known in advance. Thus, there are 4000 audit trails. By generation 9, each of these 4000 audit trails recursively incorporates information about operations involving up to 1024 ancestors. In order to minimize the size of the audit trail (which depends on the number of generations involved), we selected a relatively large population (i.e. 4000) so as to force down the number of generations needed to produce a 100% correct individual. The audit trail for the single individual of interest in generation 9 alone occupies about 27 densely-printed pages.

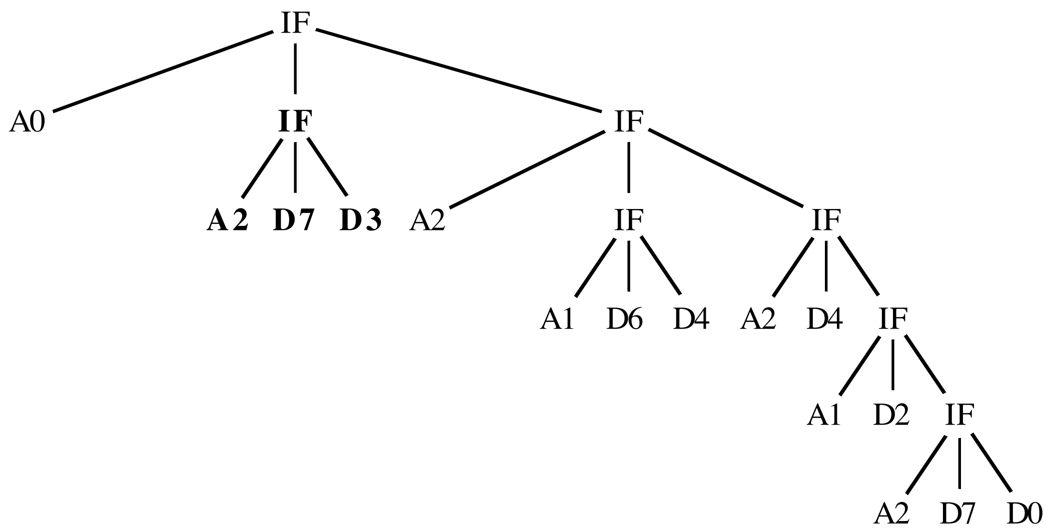
The creative role of crossover and case-splitting is illustrated by an examination of the genealogical audit trail for the l00% correct individual emerging at generation 9.

The l00% correct individual emerging at generation 9 is the child resulting from the most common genetic operation used in the process, namely crossover. The male parent from generation 8 had rank location of 58 (out of 4000) in the population and scored 1792 hits (out of 2048). The female parent from generation 8 had rank location 1 and scored 1920 hits. Note that it is entirely typical that the individuals selected to participate in crossover had relatively good rank locations in the population since crossover is performed among individuals in a mating pool created using fitness proportionate reproduction.

The male parent from generation 8 (scoring 1792) was

```
(IF A0 (IF A2 D7 D3)
       (IF A2 (IF A1 D6 D4)
              (IF A2 D4 (IF A1 D2 (IF A2 D7 D0)))))).
```

The male parent is shown below:



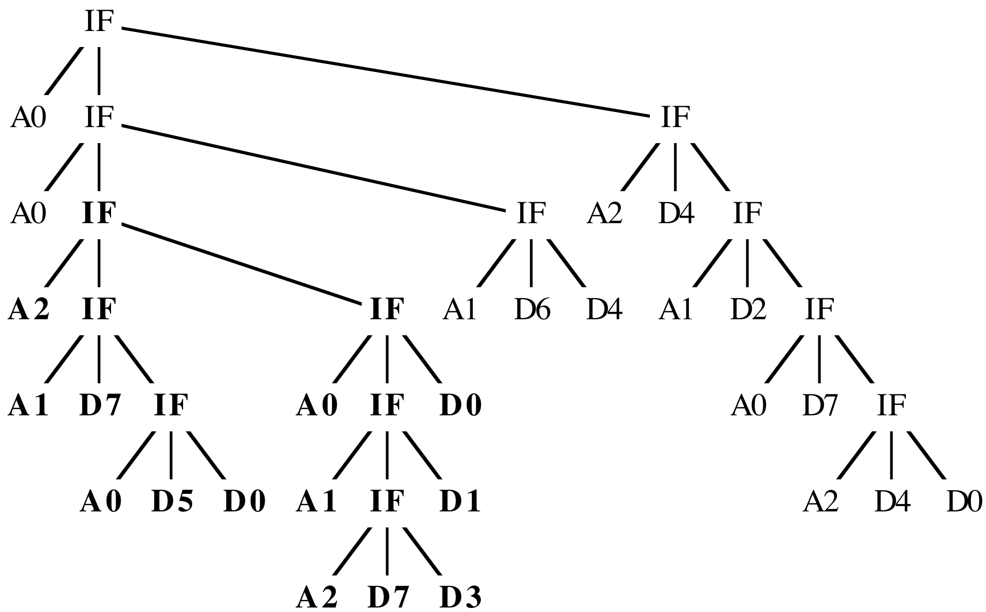
Note that this male parent starts by examining address bit A0. If A0 is T, the emboldened and underlined portion then examines address bit A2, and, partially blindly, makes the output equal D7 or D3 without even considering address bit A1. Moreover, the underlined portion of this individual does not even contain data bits D1 and D5. On the other hand, when A0 is NIL, this individual is 100% correct. In that case, it examines A2 and, if A2 is T, it then examines A1 and makes the output equal to D6 or D4 according to whether A1 is T or NIL. Moreover, if A2 is NIL, it twice retests A2 (unnecessarily, but harmlessly) and then correctly makes the output equal to `(IF A1 D2 D0)`. In other words, this imperfect individual handles part of its environment correctly and part of its environment incorrectly. In particular, this father correctly handles the even-numbered data bits and often incorrectly handles the odd-numbered data bits.

The tree representing this male parent has 22 points. The crossover point was chosen at the second occurrence of the function IF. That is, the crossover fragment consists of the incorrect, underlined sub-expression `(IF A2 D7 D3)`.

The female parent from generation 8 (scoring 1920) was

```
(IF A0 (IF A0 (IF A2 (IF A1 D7 (IF A0 D5 D0))
                     (IF A0 (IF A1 (IF A2 D7 D3) D1) D0))
              (IF A1 D6 D4))
       (IF A2 D4 (IF A1 D2 (IF A0 D7 (IF A2 D4 D0))))))
```

The female parent from generation 8 is shown below:



The tree representing this female parent has 40 points. The crossover point was chosen at the third occurrence of the function IF. That is, the crossover fragment consists of the emboldened and underlined sub-expression. This sub-expression correctly handles the case when A0 it T by making the output equal to D7 when the address bits are 111, by making the output equal to D5 when the address bits are 101, by making the output equal to D3 when the address bits are 011, and by making the output equal to D1 when the address bits are 001. This female parent does not correctly do as well when A0 is NIL. In other words, this mother correctly handles the odd-numbered data bits and incorrectly handles the even-numbered data bits.

Thus, these two imperfect individuals contain complementary, co-adapted portions which, when mated together, produce a 100% correct offspring individual. In effect, the creative effect of the crossover operation blended the two "cases" of the implicitly "case-split" environment into a single 100% correct solution.

Of course, not all such combinations of individuals that correctly handle groups of cases from the environment are useful and productive. In fact, a large fraction of the individuals produced by the genetic operations are useless. But the existence of a population of alternative disjunctive solutions to a problem provides the ingredients with which genetic recombination (crossover) can produce some improved individuals. The relentless pressure of natural selection based on fitness then causes these improved individuals to be preserved and to proliferate. Moreover, genetic variation and the existence of a population of alternative disjunctive solutions to a problem prevents the entire process from being trapped on local maxima.

Interestingly, the same crossover that produced the 100% correct individual also produced a "runt" scoring only 256 hits. In this particular crossover, the two crossover fragments not used in the 100% correct individual combined to produce an unusually unfit individual. This is one of the reasons why there is considerable variability from generation to generation in the worst single individual in the population whereas there usually is a generally monotonic improvement, from generation to generation, in both the best single individual in the population and the average fitness of the population.

As one traces the ancestry of the 100% correct created individual in generation 9 deeper back into the genealogical audit tree (i.e. towards earlier generations), one encounters parents scoring generally fewer and fewer hits. That is, one encounters more S-expressions that perform irrelevant, counterproductive, partially blind, and incorrect work. But if we look at the sequence of "hits" in the forwards direction, we see localized hill-climbing in the search space occurring in parallel throughout the population as the creative operation of crossover recombines complementary, co-adapted portions of parents to produce improved offspring.

4.1.2 THE BOOLEAN 6-MULTIPLEXER AND 3-MULTIPLEXER FUNCTIONS

We have also applied the genetic programming paradigm to the simple Boolean 6-multiplexer and 3-multiplexer. For example, in one run (Koza 1989), we obtained the following 100% correct solution of the Boolean 6-multiplexer problem:

```
(IF (AND A0 A1) D3  (IF A0 D1 (IF A1 D2 D0))).
```

Note that the result of the genetic programming paradigm is always inherently hierarchical. In addition, default hierarchies often emerge from the genetic programming paradigm. Default hierarchies incorporate partially correct sub-rules into a perfect overall procedure by allowing the partially correct sub-rules to handle the majority of the cases and by then dealing another way with certain specific cases. This solution is a default hierarchy. In this expression, the output defaults to `(IF A0 D1 (IF A1 D2 D0))`; however, in the specific case when both address bits of the 6-multiplexer problem are 11, the output is the data bit D3. Default hierarchies are considered desirable in induction problems and classifier systems (Goldberg 1989, Holland 1986, Holland et. al. 1986) because they are often parsimonious and they are a human-like way of dealing with situations. Wilson's noteworthy BOOLE experiments (1987) originally found a set of eight if-then classifier system rules for the Boolean 6-multiplexer that correctly (but exhaustively) handled each particular subcase of the problem. Subsequently, Wilson (1988) modified the credit allocation algorithm of Holland's classifier system and successfully produced a default hierarchy that more parsimoniously solved the problem. Such default hierarchies often emerge from the genetic programming paradigm.

In addition, we tried the 6-multiplexer function with the IF function deleted from the function set. That is, the function set was F = {AND, OR, NOT}. In one run, we obtained the following 100% correct solution:

```
(AND (OR A0 (OR D2 (NOT A1)))
     (OR (AND (AND (OR (AND (OR D0 A0) D1) A1) D3) D3)
         (AND (OR (AND D1 (NOT A1)) (NOT (AND A0 A0)))
              (OR (AND D0 (NOT (OR A0 (OR A1 (AND (OR (AND (OR (AND A0
A0) A1) D1) A1) D1)))))
                 (OR A1 (AND (OR (AND A0 A0) A1) D1)))))).
```

Finally, we tried the 3-multiplexer function, which is equivalent to the simple IF (i.e. if-then-else) function with the function set F = {AND, OR, NOT}. In one run, we obtained the following 100% correct solution:

```
(OR (AND (AND D0 D0) (NOT A0)) (AND D1 (AND D1 (OR D0 A0)))),
```

which is, in disjunctive normal form, equivalent to

```
(OR (AND (NOT A0) D0)
    (AND A0 D1)))).
```

4.1.3 NON-RANDOMNESS OF THESE RESULTS

The number of possible compositions of the available set of functions and terminals is very large. In particular, the number of possible trees representing these compositions of functions increases rapidly as a function of the number of points in the tree. This is true because of the large number of ways of labeling the points of a tree with functions and terminals. The number of possible compositions of functions is very large in relation to the 40,000 individuals processed in generations 0 through 9 involved in the run above.

There is a theoretic possibility that the probability of a solution to a given problem may be low in the original search space of the 11-multiplexer problem (i.e. all Boolean functions of 11 arguments), but that the probability of randomly generating a composition of functions that solves the problem might be significantly higher in the space of randomly generated compositions of functions. The Boolean 11-multiplexer function is a unique function out of the $2^{2^{11}}$ (i.e. 2^{2048}) possible Boolean functions of 11 arguments and one output. The probability of randomly choosing zeroes and ones for the 2^{11} lines of a truth table so as to create this particular Boolean function is only 1 in $2^{2^{11}}$ (i.e. 2^{2048}). However, there is a theoretic possibility that the probability of randomly generating a composition of the functions AND, OR, NOT, and IF that performs the 11-multiplexer function might be higher than 1 in 2^{2048}.

There is no *a priori* reason to believe that this is the case. That is, there is no *a priori* reason to believe that compositions of functions that solve the Boolean multiplexer problem (or any of the other problems discussed in this paper) are denser in the space of randomly generated compositions of functions than solutions to the problem in the original search space of the problem. Nonetheless, there is a possibility that this is the case even though there is no *a priori* reason to think that it is the case.

To test against this possibility, we performed the following control experiment for the Boolean 11-multiplexer problem. We generated 1,000,000 random S-expressions to check if we could randomly generate a composition of functions that solved the problem. For this control experiment, we used the same algorithm and parameters used to generate the initial random population in the normal runs of the problem. No 100% correct individual was found in this random search. In fact, the high score in this random search was only 1408 hits (out of a possible 2048 hits) and the low score was 704 hits. Moreover, only 10 individuals out of 1,000,000 achieved this high score of 1408. The high point of the histogram distribution of hits among these 1,000,000 random individuals came at 1152 hits (with l83,820 individuals); the second highest point came at 896 hits (with 168,333 individuals); and the third highest point came at 1024 hits (with 135,379 individuals).

A similar control experiment was conducted for the Boolean 6-multiplexer problem (with a search space of 2^{2^6}, i.e. 2^{64}). Since the environment for the 6-mutliplexer problem had only 64 environmental cases (as compared with 2048 cases for the 11-multiplexer), it was practical to evaluate even more randomly generated individuals (i.e. 10,000,000) in this control experiment. As before, no 100% correct individual was found in this random search. In fact, no individual had more than 52 (of 64 possible) hits. As with the 11-multiplexer, the size of the search space (2^{64}) for the 6-mutliplexer is very large in relation to the number of individuals that are processed in a typical run solving the 6-mutliplexer problem.

Additional control experiments involving between 1,000,000 and 20,000,000 individuals each were run on almost all of the other problems described in this paper without ever encountering any 100% correct solutions.

We conclude that solutions to the problems in the space of randomly generated compositions of functions are not denser than solutions in the original search space of the problem. Therefore, we conclude that <u>the results described in this paper are not the fruits of random search.</u>

As a matter of fact, we have evidence suggesting that the solutions to at least some of the problems described in this paper are appreciably <u>sparser</u> in the space of randomly generated compositions of functions than solutions in the original search space of the problem.

Consider, for example, the exclusive-or function. The exclusive-or function is the parity function with two Boolean arguments. The parity function of k Boolean arguments returns 1 if the number of arguments equal to 1 is odd and returns 0 otherwise. Whereas there are 2^{64} Boolean functions with 6 arguments and 2^{2048} Boolean functions with 11 arguments, there are only 16 (i.e. $2^{2^2} = 2^4$) Boolean functions with two Boolean arguments. That is, the exclusive-or function is one of only 16 possible Boolean functions with two Boolean arguments and one output. Thus, in the search space of Boolean functions, the probability of randomly choosing zeroes and ones for the 16 (i.e. $2^{2^2} = 2^4$) lines of a truth table that realizes this particular Boolean function is only 1 in 16.

We generated 100,000 random individuals using a function set consisting of the basic Boolean functions F = {AND, OR, NOT}. If randomly generated compositions of the basic Boolean functions that realize the exclusive-or function were as dense as solutions are in the original search space of the problem (i.e. the space of Boolean functions of 2 arguments), we would expect about about 6250 in 100,000 random compositions of functions (i.e. 1 in 16) to realize the exclusive-or function. Instead, we found that only 110 out of 100,000 randomly generated compositions that realized the exclusive-or function. This is a frequency of only 1 in 909 compared to the expected frequency of 1 in 16. In other words, randomly generated compositions of functions realizing the exclusive-or function are about 57 times <u>sparser</u> than solutions in the original search space of Boolean functions.

Similarly, we generated an additional 100,000 random individuals using a function set consisting of the basic Boolean functions F = {AND, OR, NOT, IF}. We found that only 116 out of 100,000 randomly generated compositions realized the exclusive-or function (i.e. a frequency of 1 in 862). That is, with this new function set, randomly generated compositions of functions realizing the exclusive-or problem are still about 54 times <u>sparser</u> than solutions in the original search space of Boolean functions.

In addition, we performed similar experiments on two Boolean functions with three Boolean arguments, namely, the 3-parity function and the 3-multiplexer function (i.e. the If-Then-Else function). There are only 2^{2^3} (i.e. $2^8 = 256$) Boolean functions with three Boolean arguments. We found no randomly generated composition of functions out of 3,000,000 random compositions of the functions AND, OR, and NOT that realized either the 3-parity function or the 3-multiplexer function. If the probability of randomly generating a composition of functions realizing either of these two problems was as high as the probability of randomly choosing zeroes and ones for the 2^8 lines of a truth table (i.e. 1 in 256), we would have expected about 11,719 random compositions out of 3,000,000 to realize both of these Boolean functions with three arguments.

These three results concerning the exclusive-or function (i.e. the 2-parity functions), the 3-parity function, and the 3-multiplexer functions should not be too surprising since the parity and multiplexer functions have long been identified by researchers as functions that often pose difficulties for paradigms for machine learning, artificial intelligence, neural nets, and classifier systems. This well-recognized difficulty is the reason why the parity and multiplexer functions (with various number of arguments) are often used as benchmark functions in studies of paradigms for machine learning, artificial intelligence, neural nets, and classifier systems.

In summary, as to these three benchmark Boolean functions, compositions of functions solving the problem are substantially <u>less dense</u> than solutions are in the search space of the original problem.

The reader would do well to remember the origin of the concern that compositions of functions solving a problem might be denser than solutions to the problem are in the search space of original problem. In Lenat's work on discovering mathematical laws via heuristic search (1976) and other related work (Lenat 1983), the mathematical laws being sought were stated, in many cases, directly in terms of the list, i.e. <u>the</u> primitive data type of the LISP programming language. In addition, the lists in Lenat's artificial mathematical (AM) laws were manipulated by list manipulation functions that are unique or peculiar to LISP. Specifically, in many experiments in Lenat (1976), the mathematical laws sought were stated directly in terms of lists and list manipulation functions such as, CAR (which returns the first element of a list), CDR (which returns the tail of a list), etc. In Lenat's *mea culpa* article "Why AM and EURISKO appear to work" (Lenat and Brown 1984), Lenat recognized that LISP syntax may have overly facilitated discovery of his previously reported results, namely, mathematical laws stated in terms of LISP's list manipulation functions and LISP's primitive object (i.e. the list).

In contrast, the problems described in this paper are neither stated nor solved in terms of objects or operators unique or peculiar to LISP. The solution to the Boolean multiplexer function is expressed in terms of ordinary Boolean functions (such as OR, AND, NOT, and IF). The solutions to the numerical problems discussed herein (such as symbolic regression, broom balancing) are expressed in terms of the ordinary arithmetic operations (such as addition, subtraction, multiplication, and division). The solutions to the planning problems (such as block stacking) are expressed in terms of ordinary iteration operations and various domain-specific robotic actions (such as robotic actions that move a block from one place to another). Virtually any programming language could be used to express the solutions to these problems. As previously discussed, the LISP programming language was chosen for the work reported in this paper primarily because of the many convenient features of LISP (notably the fact that data and programs have the same form in LISP and that this common form corresponds to the parse tree of a computer program) and the many convenient features of available implementations of LISP (notably the ease with which computer programs can be manipulated as data and then immediately executed). The LISP programming language was <u>not</u> chosen because of the presence in LISP of the list as a primitive data type or because of LISP's functions for manipulating lists. In fact, neither lists nor list manipulation functions are involved in <u>any</u> of the problems described in this paper (except in the irrelevant and indirect sense that the LISP programming language uses lists to do things, unseen by the user, that other programming languages do in different ways). The parse tree that LISP makes conveniently available to us for manipulation is the <u>same</u> parse tree that other programming languages construct internally at the time of compilation. This parse tree is nothing more than a direct mapping of the given composition of functions (i.e. the given computer program). We need access to this parse tree to do crossover in the way we want to do it (namely, on sub-parts of computer programs). The LISP programming language gives us this convenient access the parse tree, the ability to conveniently manipulate this program as if it were data, and the convenient ability to immediately execute a newly created parse tree.

In summary, there is no *a priori* reason (nor any reason we have since discovered) to think that there is anything about the syntax of the programming language we chose to use here (i.e. LISP) that makes it easier to discover solutions to problems involving ordinary (i.e. non-list) objects and ordinary (i.e. non-list) functions. In addition, the control experiments verify that the results obtained herein are not the fruits of a random search.

4.2. PLANNING

Planning in artificial intelligence and robotics requires finding a plan that receives information from sensors about the state of the various objects in a system and then uses that information to select a sequence of actions to change the state of the objects in that system.

4.2.1. BLOCK STACKING

Nilsson (1988a) has presented a robotic action network that solves a planning problem, described to Nilsson (1988b) by Ginsberg, involving rearranging uniquely labeled blocks in various towers from an arbitrary initial arrangement into an arbitrary specified new order on a single target tower. In the version of the problem involving 9 blocks, the blocks are labeled with the 9 different letters of "FRUITCAKE" or "UNIVERSAL." In the experiment here, the goal is to automatically generate a plan (Nilsson 1980) that solves this problem.

This problem is typical of many problems in artificial intelligence in that it is primarily symbolic. This problem illustrates the technique of associating LISP variable atoms with the state variables of a problem and of using functions to cause side effects altering the state of a system. This problem is also typical of computer programs in general in that an iterative operator DU ("Do Until") is used in the solution of the problem. In the discussion below, we follow the formulation of the problem and the specific lists, sensors, and functions described by Nilsson, although, there exist many equivalent and alternative versions of this problem.

Three lists are involved in the formulation of the problem. The GOAL-LIST is the ordered set specifying the desired final order in which the blocks are to be stacked in the target tower (i.e. "UNIVERSAL"). The STACK is the ordered set of blocks that are currently in the target tower (where the order is important). The TABLE is the set of blocks that are currently not in the target tower (where the order is not important). The initial configuration consists of certain blocks in the STACK and the remaining blocks on the TABLE. The desired final configuration consists of all the blocks being in the STACK in the order specified by GOAL-LIST and no blocks being on the TABLE .

Three sensors dynamically track the system in the formulation of the problem as presented by Nilsson and Ginsberg. The sensor TB (“Top correct Block”) dynamically specifies the CAR (i.e. first element) of the list which is the longest CDR (i.e. list of remaining elements) of the list STACK that matches a CDR of GOAL-LIST. The sensor NN (“Next Needed”) dynamically specifies the next needed block for the STACK (i.e. the immediate predecessor of TB in GOAL-LIST). The sensor CS dynamically specifies the CAR of the STACK (i.e. the top block of the STACK).

Thus, the set of terminals available for solving this problem is T = {TB, NN, CS}. Each of these terminals is a variable atom that may assume, as its value, one of the 9 block labels or NIL.

The combined set of functions available for solving the problem here contains 5 functions F = {MS, MT, DU, NOT, EQ}. The functions NOT and EQ are the usual Boolean Common LISP negation and equality functions. The other three functions are described below.

The function MS (“Move to the Stack”) has one argument. The S-expression (MS X) moves block X to the top of the STACK if X is on the TABLE. This function MS does nothing if X is already on the STACK, if the table is empty, or if X itself is NIL. Both this function and the function MT described below returns NIL if they do nothing and T if they do something; however, their real functionality is their side effects on the STACK and TABLE, not their return values.

The function MT (“Move to the Table”) has one argument. The S-expression (MT X) moves the top item of the STACK to the TABLE if the STACK contains X anywhere in the STACK. This function MT does nothing if X is on the TABLE, if the STACK is empty, or if X itself is NIL.

The iterative operator DU ("Do Until") has two arguments. The S-expression (DU WORK PREDICATE) iteratively does the WORK until the PREDICATE becomes satisfied (i.e. becomes T). The DU operator is similar to the “REPEAT...UNTIL” loop found in many programming languages. Note that the WORK and PREDICATE arguments are not evaluated outside the iterative DU operator and then passed to the DU operator when the DU operator is called. Instead, these arguments must be evaluated dynamically inside the DU operator on each iteration. First, the WORK is evaluated inside the DU operator. Then the PREDICATE is evaluated inside the DU operator. These two separate evaluations are performed, in sequence, using the LISP function EVAL inside the DU operator. Note that in an iterative construction, the execution of the WORK will almost always change some variable that will then be tested by PREDICATE. Indeed, that is usually the purpose of the loop. Thus, it is important to suppress premature evaluation of the WORK and PREDICATE arguments of the DU operator. The evaluation of arguments to the other iterative and summation operators described elsewhere in this article must be similarly postponed. Although not used in this problem, the iterative function DU, in general, has an indexing variable II which is updated for each iteration.

Because the genetic programming paradigm described herein involves executing randomly generated computer programs, a number of computer implementation issues must be addressed. In particular, individual S-expressions in the genetic population will often contain an unsatisfiable termination predicate. Thus, it is a practical necessity (when working on a serial computer) to place a limit on the number of iterations allowed by any one execution of a DU operator. Moreover, since the individual S-expressions in the genetic population often contain complicated and deep nestings of numerous DU operators, a similar limit must be placed on the total number of iterations allowed for all DU functions that may be evaluated in the process of evaluating any one individual S-expression for any particular environmental case. Thus, the termination predicate of each DU operator is actually an implicit disjunction of the explicit predicate argument PREDICATE and two additional implicit termination predicates. The typical "time out" limits that we have used are that the DU operator "times out" if there have been more than 25 iterations for an evaluation of a single DU operator or if there have been a total of more than 100 iterations for all DU operators that are evaluated for a particular individual S-expression for a particular environmental case. Of course, if we could execute all the individual LISP S-expressions in parallel (as nature does) so that the infeasibility of one individual in the population does not bring the entire process to a halt, we would not need these limits. Note that even when a DU operator times out, it nevertheless returns a value. In particular, the DU operator evaluates to T unless one of the two implicit termination predicates times out. The value resulting from this evaluation of the DU operator is, of course, in addition to the side effects of the DU function on the state variables of the system (particularly the STACK and TABLE in the block-stacking problem). If the predicate of a DU operator is satisfied when the operator is first called, then the DU operator does no work at all and simply returns a T.

Note that the fact that each function returns some value under all conditions (in addition to whatever side effects it has on the STACK and TABLE) and the inherent flexibility of the LISP language guarantee that every possible individual S-expression can be executed and evaluated for any composition of functions and arguments that may arise.

The environment consists of millions of different environmental cases of N blocks distributed between the STACK and the TABLE. The raw fitness of a particular individual plan in the population is the number of environmental cases for which the particular plan produces the desired final configuration of blocks after the plan is executed.

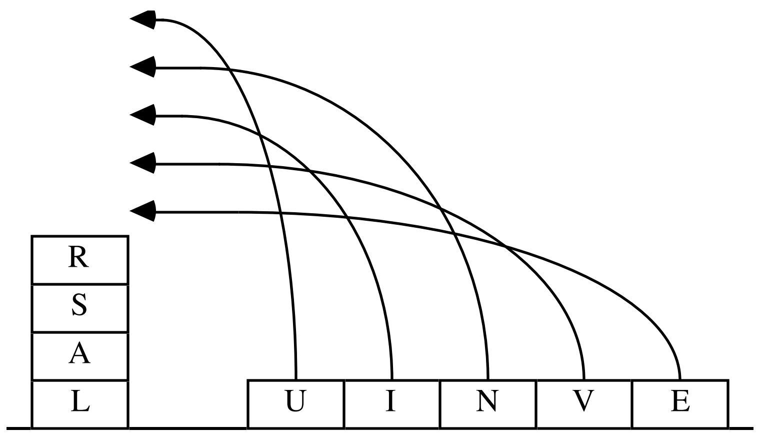
The computation of fitness in this problem (and, indeed, in many genetic algorithm and adaptive systems problems) can be significantly shortened by consolidating various inherently similar initial configurations or by sampling. In particular, if there are N blocks, there are N+1 cases in which the blocks, if any, in the initial STACK are all in the correct order and in which there are no out-of-order blocks on top of the correctly-ordered blocks in the initial STACK. There are also N-1 additional cases where there is precisely one out-of-order block in the initial STACK on top of whatever number of correctly-ordered blocks, if any, happen to be in the initial STACK. There are additional initial cases with more than one out-of-order blocks in the initial STACK on top of various number of correctly-ordered blocks in the initial STACK. In lieu of an environment of up to several million environmental cases, we constructed an environment consisting of (1) the l0 cases where the 0-9 blocks in the STACK are already in correct order, (2) the 8 cases where there is precisely one out-of-order block in the initial STACK on top of whatever number of correctly-ordered blocks, if any, happen to be in the initial STACK, and (3) a structured random sampling of 148 additional environmental cases with 0, 1, 2, ...,8 correctly-ordered blocks in the initial STACK and various random numbers 2, 3, 4, ... out-of-order blocks on top of the correctly-ordered blocks. The complete structured random sampling used for this problem contained a total of 166 environmental cases so that raw fitness ranged over 167 values between 0 and 166. Obviously, this consolidation and sampling process must be done with some care so that the process is not misled into producing solutions that correctly handle the smaller environment and do not correctly handle the entire environment.

4.2.2. CORRECTLY STACKING BLOCKS

The first version of the block-stacking problem involves finding a general plan which can correctly stack the 9 blocks onto the STACK in the desired order after starting with any of the 166 environmental cases. Each plan is executed (evaluated) once for each of the 166 environmental cases.

The initial random population of plans has predictably low fitness. Many of these initial random plans were complicated, inefficient, pointless, or counter-productive. Typical random initial plans are plans such as (EQ (MT CS) NN) and (MS TB). This first plan unconditionally moves the top of the STACK to the TABLE and then performs the useless Boolean comparison between the sensor value NN and the return value of the MT function. The second plan (MS TB) futilely attempts to move the block TB (which already is in the STACK) from the TABLE to the STACK. Many initial random plans are so ill-formed that they perform no action at all on the STACK and the TABLE. Since these leave untouched the one environmental case consisting of an already perfectly arranged, these plans achieve a raw fitness level of 1 (out of a maximum of 166) . Many other initial random plans are even more unfit and even disrupt a perfectly arranged initial STACK. These plans achieve a fitness level of 0. Some initial random plans achieve modest fitness levels such as 2, 3 or 4, because they contain particular specific action sequences that happen to work on a specific small number of the environmental cases. For example, the plan `(EQ (MS NN) (EQ (MS NN) (MS NN)))` moves the next needed block (NN) from the TABLE to the STACK three times. This plan works in the 4 particular specific environmental cases where the initial STACK consists of 6, 7, 8, or 9 correct blocks and no out-of-order blocks.

In one run, an individual plan emerged in generation 5 that correctly handled 10 of the 166 environmental cases. This plan correctly handles the 10 cases in group (1) above where the blocks, if any, initially on the STACK happen to already be in the correct order and in where there are no out-of-order blocks on top of these correctly-ordered blocks. The plan `(DU (MS NN) (NOT NN))` uses the iterative operator DU to iteratively do the work (MS NN) of moving the needed block onto the STACK from the TABLE until the predicate (NOT NN) is satisfied. This predicate is satisfied when there are no more blocks needed to finish the STACK (i.e. the "next needed" sensor NN is NIL). This plan, of course, does not produce a correct final STACK if any block initially on the STACK was incorrect. The figure below shows this partially correct plan moving 5 needed blocks (E, V, N, I, and U) to a STACK already containing R, S, A, and L in correct order.

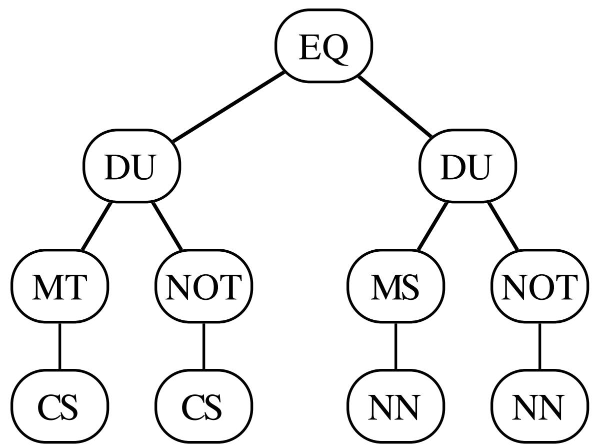


Although this plan performs incorrectly in 156 of the 166 environment cases, it will prove to be be a useful "building block" in the final 100% correct plan.

As additional generations are run, the performance of the best single individual plan in the population typically increases somewhat from generation to generation and correctly deals with a few more additional cases in the environment. At the same time, the overall average fitness of the population also tends to increase somewhat from generation to generation as the population begins to contain additional higher scoring plans. In one run, the best individual in the population in generation 10 achieved a perfect score (that is, the plan produced the desired final configuration of blocks in the STACK for 100% of the initial environmental cases). This 100% correct plan

```
(EQ (DU (MT CS) (NOT CS)) (DU (MS NN) (NOT NN))).
```

is depicted graphically below:



This plan consists of two sub-plans which are connected via the function EQ (which is merely serving as a connective). The first sub-plan `(DU (MT CS) (NOT CS))` iteratively does the work of moving CS (i.e. the top of the STACK) to the TABLE until the predicate `(NOT CS)` becomes T (i.e. the top of the STACK becomes NIL). The second sub-plan does the work of iteratively moving the next needed block NN to the STACK until there is no remaining next needed block NN.

Notice that the previously discovered, partially correct plan `(DU (MS NN) (NOT NN))` became incorporated as a subplan into the final 100% correct solution. This subplan became part of a hierarchy created as a result of crossover applied in proportion to fitness.

4.2.3. EFFICIENTLY STACKING BLOCKS

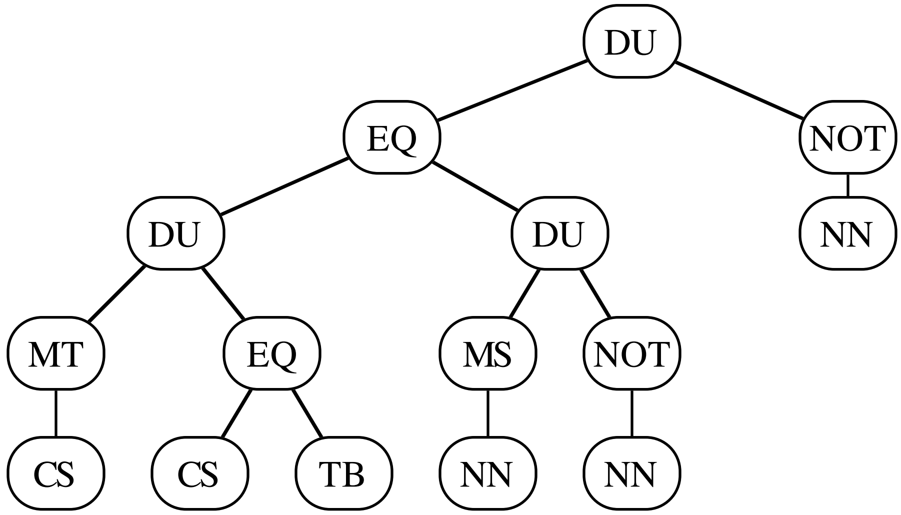
The particular 100% correct solution discovered above was typical of the vast majority of plans generated in that it inefficiently removed all the blocks from the STACK (even if they were already in the correct order) and then moved the blocks, one by one, from the TABLE to the STACK. As a result, this plan used 2319 block movements to handle the 166 environmental cases. The most efficient way to solve this problem, in terms of minimizing total block movements, is to remove only the out-of-order blocks from the STACK and to then move the next needed blocks to the STACK. As it happens, only 1641 block movements are required over the 166 environmental cases if the most efficient approach is used.

We can simultaneously breed the population for two attributes at once. In particular, we can specifically breed a population of plans to stack blocks for <u>both</u> correctness and efficiency by using a combined fitness measure that assigns a majority of the weight (say 75%) to correctness and a minority of weight (say 25%) to a secondary attribute (i.e. efficiency).

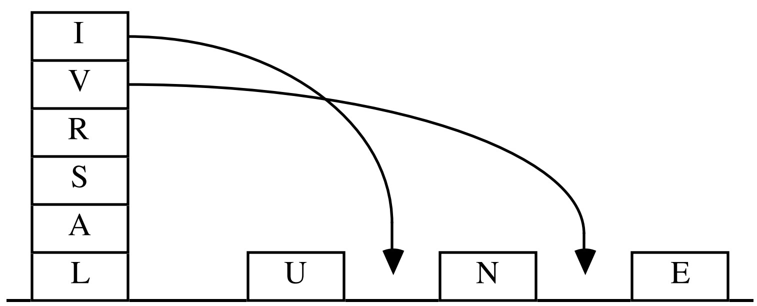
In one run, for example, the best individual from the initial random population (generation 0) performed correctly in only 1 of the 166 environmental cases and involved a total of 6590 block movements. However, by generation 11, the best individual in the population was

```
(DU (EQ (DU (MT CS) (EQ CS TB))
        (DU (MS NN) (NOT NN)))
    (NOT NN))
```

This plan is l00% correct and 100% efficient in terms of total block movements. It uses the minimum number (1641) of block movements to correctly handle all 166 environmental cases. This plan is graphically depicted below:



In this plan, the sub-plan `(DU (MT CS) (EQ CS TB))` iteratively moves CS (the top block) of the STACK to the TABLE until the predicate (EQ CS TB) becomes satisfied. This predicate becomes satisfied when CS (the top of the stack) equals TB (top correct block in the STACK). The figure below shows the out-of-order blocks (I and V) being moved to the TABLE from the STACK until R becomes the top of the STACK. When R is the top of the STACK, CS equals TB.



Then, the previously discovered second sub-plan `(DU (MS NN) (NOT NN))` iteratively moves the next needed blocks (NN) to the STACK until there is no longer any next needed block.

Notice that the function EQ serves only as a connective between the two sub-plans. Notice also that the outermost DU function performs no function (but does no harm) since the predicate `(NOT NN)` is satisfied at the same time as the identical predicate of the second sub-plan. In that regard, the functionless elements are similar to the approximately 99% of nucleotide bases (out of approximately 2.87 billion) in a molecule of human deoxyribonucleic acid that never get expressed into protein.

4.2.4. A PARSIMONIOUS EXPRESSION FOR STACKING BLOCKS

The particular solution discovered above is 100% correct and 100% efficient, but there are 15 points in the tree corresponding to the S-expression. By including parsimony in the fitness function with 25% weight (in addition to correctness with 75% weight), we bred the population for parsimony and correctness and obtained the following S-expression consisting of only 12 points:

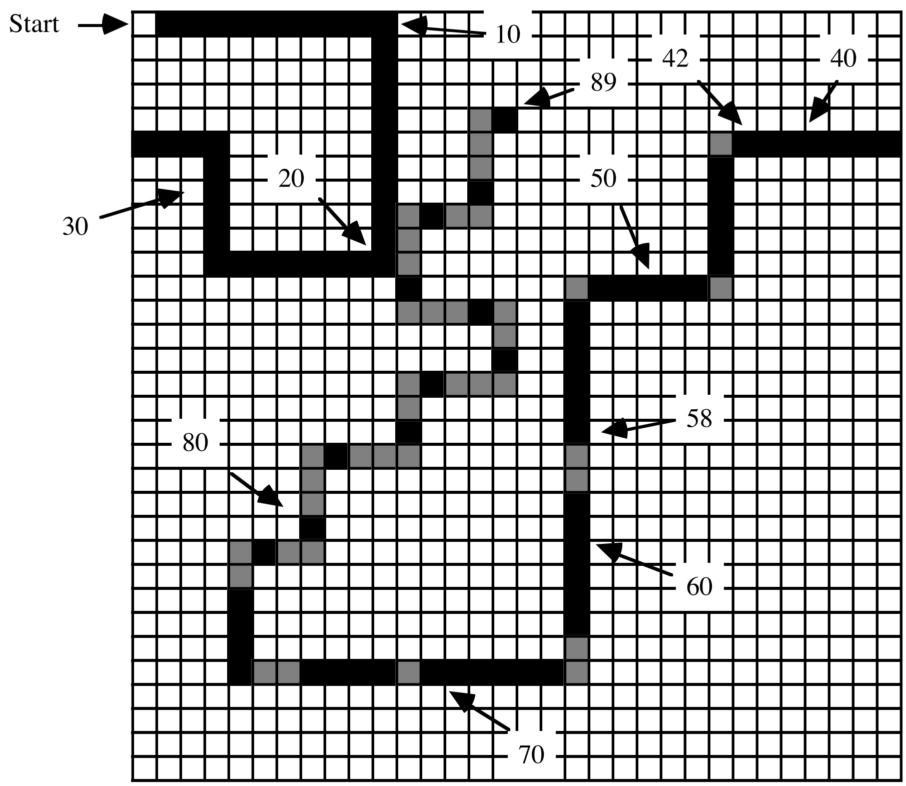
```
(EQ (DU (MT CS) (EQ CS TB))
    (DU (MS NN) (NOT NN))).
```

4.2.5. ARTIFICIAL ANT - TRAVERSING THE JOHN MUIR TRAIL

Jefferson, Collins et. al. (1990) have devised a complex task for an "artificial ant" attempting to traverse a trail and produced a noteworthy solution in the form of both a finite state automaton and a multi-layer recurrent neural net for directing the ant's motions.

The setting for the problem, as defined by Jefferson, Collins et. al., is a square 32 by 32 grid containing stones in 89 of the 1024 cells and nothing in the remaining cells. The "John Muir" trail is a winding trail of stones with single missing stones, double missing stones, a missing stone at some corners, double missing stones at some corners (knight moves), and triple missing stones at some corners (long knight moves). The "Santa Fe" trail (devised by Doyne Farmer) is a somewhat more complicated trail with these same features.

The "artificial ant" begins at the cell identified by the coordinates (0,0) and is facing in a particular direction (e.g. east) at the beginning of the trail. The artificial ant has a sensor that can see only the single adjacent cell in the direction the ant is currently facing. At each time step, the ant has the capacity to execute any of four operations, namely, to move forward in the direction it is facing, to turn right (and not move), to turn left (and not move), or to do nothing. The grid is toroidal so that if the ant moves off the edge of the grid, it reappears and continues on the opposite edge. The objective of the ant is to traverse the entire trail. As the ant moves into a particular cell with a stone, that stone is credited to the ant's account and the cell is then converted into a blank cell so that it is not counted again. The ant's score is measured by the total number of stones it finds within a certain time limit (200 for Jefferson, Collins et. al.).



Jefferson, Collins et. al. started by assuming that the finite automaton necessary to solve the problem would have 32 or fewer states. They then represented an individual in their population of automata by a binary string representing the state transition diagram (and its initial state) of the individual automaton. The ant's sensory input at each time step was coded as one bit and the output at each time step was coded as two bits (for the four possible operations). The next state of the automaton was coded with 5 bits. The complete behavior of an automaton could thus be specified with a genome consisting of a binary string with 453 bits (64 substrings of length 7 representing the state transitions plus 5 additional bits representing the initial state of the automaton). The finite automaton was executed at each time step. They then processed a population of 65,536 individual bit strings of length 453 on the Connection Machine using a genetic algorithm using crossover and mutation operating on a selected (relatively small) fraction of the population. After 200 generations in a particular run (taking about 10 hours on the Connection Machine), Jefferson, Collins et. al. reported that a single individual in the population emerged which attained a perfect score of 89 stones. As it happened, this single individual completed the task in exactly 200 operations.

Jefferson, Collins et. al. were similarly successful in discovering a multi-layer recurrent neural net for this task. In that effort, they assumed that the neural net necessary to solve the problem would have two processing units in the input layer (representing the two possible sensory inputs of the ant), five processing units with 7 inputs each in the hidden layer, and four processing units with 7 inputs each in the output layer (for the four possible operations). The genome for encoding the neural net contained 520 bits representing the weights associated with the inputs, the thresholds, and the initial activation levels of the processing units.

In our approach to this task using the genetic programming paradigm, the function set consisted of the functions F = {IF-SENSOR, PROGN}. The IF-SENSOR function has two arguments and evaluates the first argument if the ant's sensor senses a stone or, otherwise, evaluates the second argument. The PROGN function is the Common LISP connective function that sequentially evaluates (executes) its arguments as a "program."

The terminal set T = {ADVANCE, TURN-RIGHT, TURN-LEFT} consists of three "functions" that have no arguments and operate via their side effects on the ant's state (i.e. the ant's position on the grid or the ant's facing direction). Notice that it would have been equally acceptable to place these three "terminals" in the function set F (each with zero arguments) rather than in the terminal set T.

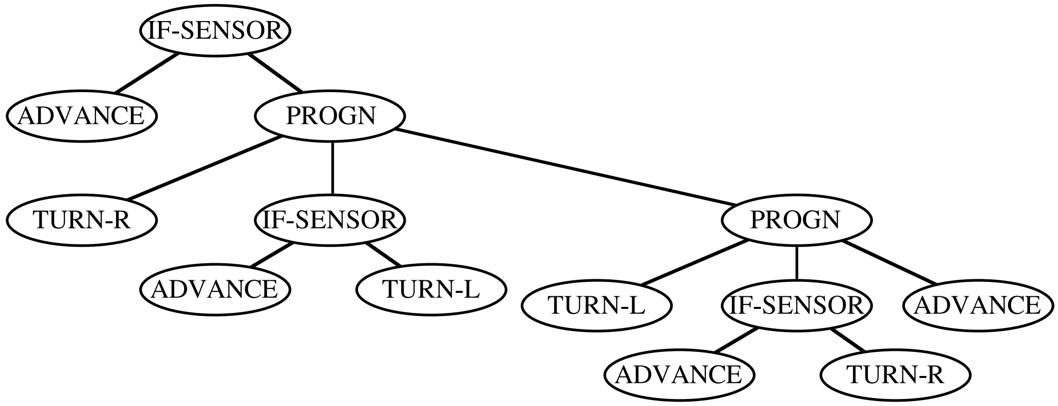
Notice that IF-SENSOR, ADVANCE, TURN-RIGHT, and TURN-LEFT correspond directly to the operators defined and used by Jefferson, Collins et. al. The "John Muir" trail was used first. We allowed 400 time steps before timing out.

The randomly generated individuals for generation 0 were, as usual, highly unfit in performing the task. Many of the individuals were blind in that they did not even consult the ant's sensor to decide what to do next. In several runs, the best randomly generated individual was an individual such as (IF-SENSOR (ADVANCE) (TURN-RIGHT)), (IF-SENSOR (ADVANCE) (TURN-LEFT)), or the equivalent. These individuals scored 42 stones (out of a possible 89). These individuals can successfully handle both right and left turns in the trail provided that a stone is not missing at the point where the trail turns. Thus, they score 42 stones, but no more, because a stone is missing where the trail turns (in the upper right area of the trail) and the ant languishes at that point (the stone behind it having been erased at time step 41).

In one run (involving our usual population size of 300 individuals), an individual scoring 89 out of 89 emerged on the 7th generation, namely,

```
(IF-SENSOR (ADVANCE)
        (PROGN (TURN-RIGHT)
               (IF-SENSOR (ADVANCE) (TURN-LEFT))
               (PROGN (TURN-LEFT)
                      (IF-SENSOR (ADVANCE) (TURN-RIGHT))
                      (ADVANCE)))).
```

This plan is graphically depicted below:



This individual plan moves the ant forward if a stone is sensed. Otherwise it turns right and then moves the ant forward if a stone is sensed but turns left (returning to its original orientation) if no stone is sensed. Then it turns left and moves forward if a stone is sensed but turns right (returning to its original orientation) if no stone is sensed. If the ant originally did not sense a stone, the ant moves forward unconditionally as its fifth operation. Note that there is no testing of the backwards directions.

The same experiment was then repeated using the somewhat more complicated "Santa Fe" trail (designed by J. Doyne Farmer) and successfully produced an equivalent plan.

4.3. SYMBOLIC FUNCTION IDENTIFICATION

Problems in the area of symbolic function identification require finding a function, in symbolic form, that fits a given sampling of data points.

4.3.1. SEQUENCE INDUCTION

Sequence induction requires developing a computational procedure that can generate any arbitrary element in an infinite sequence $S = S_0, S_1, ..., S_j, ...$ after seeing only a relatively small number of specific examples of the values of the sequence. For example, one might easily induce the computational procedure j^2+1 given only the six sequence values 1, 2, 5, 10, 17, 26. On the other hand, it is considerably harder to identify the computational procedure for S = 1, 1, 2, 3, 3, 4, 5, 5, 6, 6, 6, 8, 8, 8, 10, 9, 10, 11, 11, 12. In any case, induction is at the heart of learning and the ability to correctly perform induction is widely accepted as a means for measuring human intelligence.In this section, the problem is to induce the computational procedure (i.e. LISP S-expression) for a sequence after seeing only a small number of examples of the sequence. Both examples of sequence induction discussed below use recursion.

4.3.1.1. FIBONACCI SEQUENCE

The elements s_i of the Fibonacci sequence can be computed using the recursive expression $s_{j-1} + s_{j-2}$ (where s_0 and s_1 are both 1). Thus, in this experiment, the environment in which adaptation is to take place consists of the first 20 elements of the actual Fibonacci sequence, namely S = 1, 1, 2, 3, 5, 8, 13, 21, 34, 55,..., 4181, 6765.

The LISP S-expression

```
(+ (σ (- J 1)) (σ (- J (+ 1 1))))
```

is one possible LISP S-expression that correctly computes the Fibonacci sequence starting with J=2. In this particular representation, J is the index for the sequence element being computed. The first two elements of the sequence are given as base cases. A given S-expression can call on itself (in a true recursive way) via the function σ. . The expression (σ I) returns the value of the current S-expression for sequence position I provided I is between 0 and J-1; otherwise, it returns 0.

Notice that the σ function returns a value computed by the current S-expression, not the actual correct value of the Fibonacci sequence.

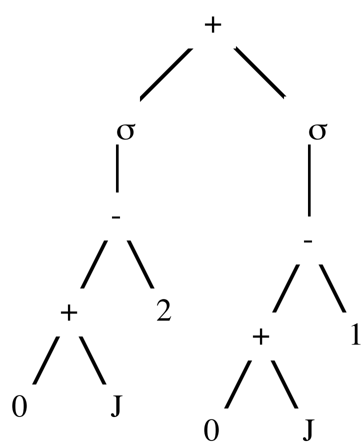
The combined set C of functions and terminals available for this problem is C = {+, -, σ, J, 0, 1, 2} having 2, 2, 1, 0, 0, 0, and 0 arguments, respectively. Note that the set F of available function must include the sequence referencing function σ (or some functional equivalent of it) because it is known that previous sequence elements are necessary in order to compute the Fibonacci sequence in the integral domain.

Examples of the random S-expressions in the initial random population for this problem included S-expressions such as `(+ 0 (+ 1 1))`, `(* 0 (- J 1))`, and `(* (* J J) (* 1 J))`. The first such S-expression, `(+ 0 (+ 1 1))`, attempts to compute the J-th element of the sequence without even considering J. The S-expression `(* 0 (- J 1))` evaluates to a constant (i.e. 0). The S-expression `(* (* J J) (* 1 J))` attempts to compute the Fibonacci sequence without making use of the essential sequence referencing function σ.

In generation 17, the best individual S-expression in the population had a perfect score of 20 matches. That individual was a 100% correct, albeit non-parsimonious, solution to the problem, namely

```
(+ (σ (- (+ 0 J) 2)) (σ (- (+ 0 J) 1))).
```

This individual is graphically depicted below:



An alternative approach to the induction of the Fibonacci sequence (in which the values of the two base cases are discovered using a σ function with 2 arguments) is presented in Koza (1989). We have also successfully performed the induction of the Fibonacci sequence (in which the values of the two base cases are discovered using a function set that included an IF function).

In order to save computer time, the elements in the sequence are computed in order for J = 2, 3, 4, ... so that previous elements of the sequence are available when an S-expression attempts to compute a new element of the sequence by recursively calling on earlier elements of the sequence.

4.3.1.2. HOFSTADTER SEQUENCE

The elements s_i of the Hofstadter sequence (Wolfram 1988) can be computed using the recursive expression

$$S_j = S_{j-s_{j-2}} + S_{j-s_{j-1}}$$

(where s_0 and s_1 are both 1). Thus, in this experiment, the environment in which adaptation is to take place consists of the first 20 elements of this highly irregular, non-monotonic Hofstadter sequence, namely S = 1, 1, 2, 3, 3, 4, 5, 5, 6, 6, 6, 8, 8, 8, 10, 9, 10, 11, 11, 12.

In one run, the number of hits was 4, 4, 3, 9, and 18 for generations 0 through 4, respectively, as the following 100% correct S-expression emerged on the generation 4:

```
(+ (σ (- J (σ  (- J 2))))  (σ  (- J (σ (- J (- 2 1))))))
```

4.3.2. SYMBOLIC REGRESSION

In linear regression, one is given a set of values of various independent variable(s) and the corresponding values for the dependent variable(s). The goal is to discover a set of numerical coefficients for a linear combination of the independent variable(s) which minimizes some measure of error (such as the sum of the squares) between the given values and computed values of the dependent variable(s). Similarly, in quadratic regression, the goal is to discover a set of numerical coefficients for a quadratic expression which similarly minimizes error. In Fourier “regression”, the goal is to discover a set of numerical coefficients for Sin and Cos functions of various periodicities which similarly minimizes error.

Of course, it is left to the researcher to decide whether to do a linear regression, quadratic regression, or a higher order polynomial regression or whether to try to fit the data points to some non-polynomial family of functions (e.g. sines and cosines of various periodicities, etc.). But, often, <u>the</u> issue is deciding what type of function most appropriately fits the data, not merely computing the numerical coefficients after the type of function for the model has already been chosen. In other words, the problem is <u>both</u> the discovery of the correct functional form that fits the data and the discovery of the appropriate numeric coefficients. We call such problems "symbolic regression."

4.3.2.1. SIMPLE EXAMPLE OF SYMBOLIC REGRESSION

For example, suppose we are given a sampling of the numerical values from an unknown curve over 20 points in some domain, such as the interval [-1, +1]. That is, we are given 20 pairs of points (x_i, y_i). These points might include pairs such as (-0.4, -0.2784), (0.5, 0.9375). (0.25, 0.33203), etc. The goal is to find a function, in symbolic form, that is a perfect fit or good fit to the 20 pairs of numerical data points. (The unknown curve happens to be $x^4+x^3+x^2+x$ for this example).

The solution to this problem of finding a function in symbolic form that fits a given sample of data can be viewed as a search for a function from a hyperspace of functions that can be composed from a set of candidate functions and arguments. The set of terminals for this particular problem consists of just the independent variable X. That is T = {X}. The set of available functions might include addition (+), subtraction (-), multiplication (*), the restricted division function (%), the sine function SIN, the cosine function COS, the exponential function EXP, and the restricted logarithm function RLOG. The restricted division function returns zero when division by zero is attempted and is used so that any composition of functions from the function set and terminals from the terminal set always produces a valid result. The restricted logarithm function RLOG returns 0 for an argument of 0 and otherwise returns the logarithm of the absolute value of the argument. These two functions are restricted in this manner so that they always return a valid floating point number. The function set is therefore F = {+, -, *, %, SIN, COS, EXP, RLOG}.

In one run, the worst single individual S-expression in the initial random population (generation 0) was

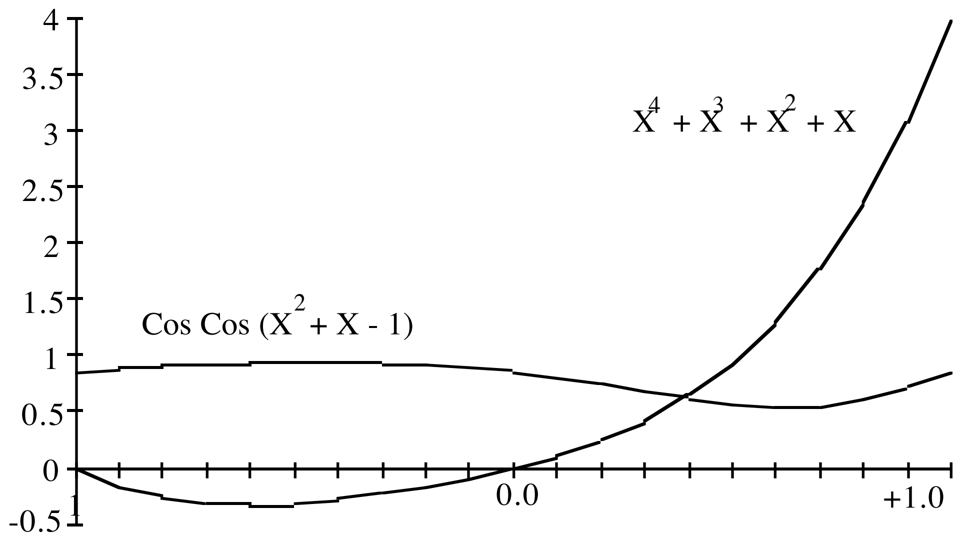
```
(EXP (- (% X (- X (RSIN X))) (RLOG (RLOG (* X X))))).
```

The sum of the absolute values of the differences between this worst single S-expression and the 20 data points was approximately 10^{23}.

The median (150th best) individual in the initial random population was

```
(COS (COS (+ (- (* X X) (% X X)) X))).
```

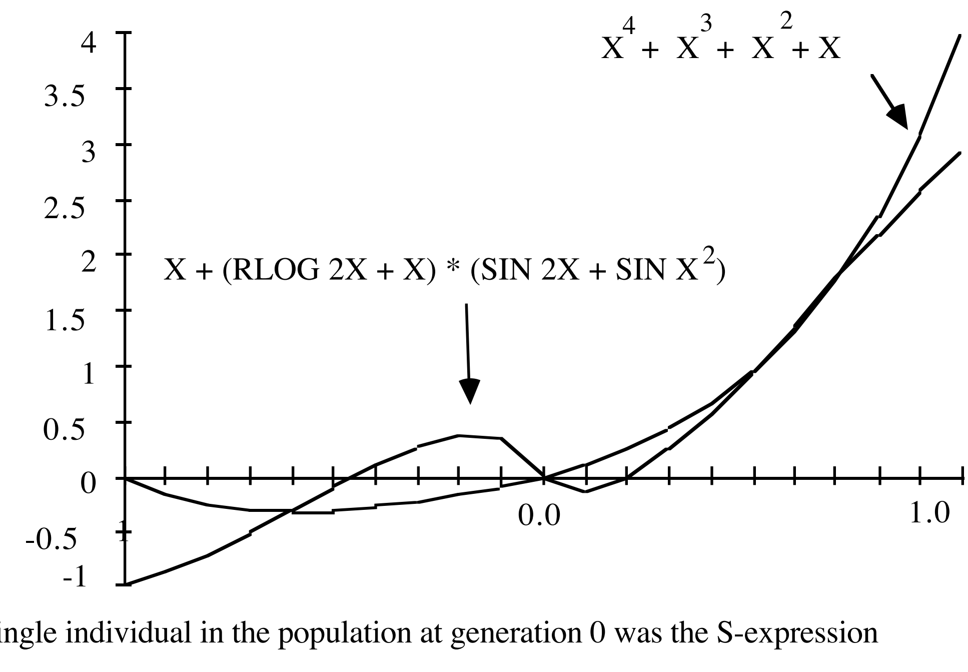
This individual is equivalent to Cos [Cos (x2 + x -1)] and has a raw fitness of this was 23.67. That is, the average distance between the curve for this individual function and the curve for $x^4+x^3+x^2+x$ for the 20 points is about 1.2.



The second best individual in the initial random population was

```
x + [RLog 2x + x] * [Sin 2x + Sin x^2]
```

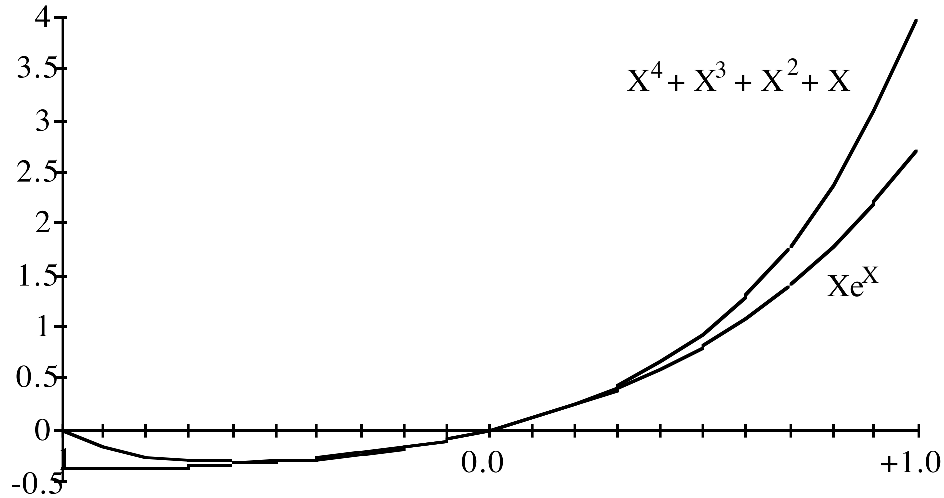
This individual has a raw fitness of 6.05. That is, the average distance between the curve for this function and the curve for $x^4+x^3+x^2+x$ for the 20 points is about 0.3.



The best single individual in the population at generation 0 was the S-expression

```
(* X (+ (+ (- (% X X) (% X X)) (SIN (- X X))) (RLOG (EXP (EXP X))))).
```

This S-expression is equivalent to xe^X. The sum of the absolute value of the differences between this S-expression and the 20 data points was 4.47 (i.e. an average difference of about 0.23 per data point).



It came within the criterion (1%) of the data in only 2 of the 20 cases. That is, it scored 2 "near hits" with the given data. Note that this number of "near hits" is used only externally for purposes of monitoring and describing the process; it is not used by the genetic algorithm.

Although xe^X is not a particularly fit (much less a perfect fit) to $x^4+x^3+x^2+x$, it is nonetheless better than the worst individual. It is better than the median individual. And, it is better than all the other 299 randomly created S-expressions. When graphed, it bears some similarity to the target S-expression. In addition, it has some of the gross characteristics of the target function. For example, both xe^X and $x^4+x^3+x^2+x$ are zero when X is zero. Also, when X approaches +1.0, xe^X approaches 2.718 while $x^4+x^3+x^2+x$ approaches the somewhat nearby value of 4.

By generation 2, the best single individual improved to

```
(+ (* (* (+ X (* X (* X (% (% X X) (+ X X)))))
         (+ X (* X X))
     X)
   X)
```

which is equivalent to $x^4 + 1.5x^3 + 0.5x^2 + x$. The raw fitness of this best individual improved to 2.57. It scored 5 "near hits." This S-expression bears greater similarity to the target S-expression than its predecessors. It is, for example, a polynomial of the correct order (i.e. 4). Moreover, the coefficients of two of the four terms are correct and the incorrect coefficients (1.5 and 0.5) are not too different from the correct coefficients (1.0 and 1.0).

Notice that even though no numerical coefficients were explicitly provided in the terminal set, the fractional coefficient 0.5 was created by the process by first creating 1/2X (by dividing X/X = 1 by X+X = 2X) and then multiplying by X. The coefficient 1.5 was similarly created.

By generation 4, the raw fitness of the best single individual in the population attained the perfect value of 0.0. This individual also scored 20 "near hits." This individual was

```
(+ X (* (+ X (* X (+ X (* X X)))) X)).
```

This S-expression is equivalent to $x^4+x^3+x^2+x$.

4.3.2.2. SYMBOLIC REGRESSION WITH CONSTANT CREATION

Discovery of the appropriate numeric coefficients is a problem that must be addressed in order to successfully do symbolic regression in general. In the foregoing simple example of symbolic regression, the terminal set consisted only of the independent variable X. There was no explicit facility for "constant creation," although the constant 1 was created by the algorithm by dividing X by X. We now address the issue of "constant creation."

The problem of "constant creation" can be solved by extending the terminal set by one ephemeral element (called “R”) during the generation of the initial random population. Thus, the terminal set would be enlarged for this particular problem and become T = {X, R}. Whenever the ephemeral atom "R" is chosen for any point of the tree during the generation of the initial random population, a random number in a specified range is generated and attached to the tree at that point. In a real-valued problem such as this problem, the random constants were floating point numbers between -1.0 and +1.0. In a problem involving integers (e.g. induction of a sequence of integers), random integers over a specified range are generated for the ephemeral “R” atoms. In a Boolean problems, the ephemeral “R” atoms are replaced by either T (True) or NIL. Note that this random generation is done anew for each point where the ephemeral “R” atom is encountered so that the initial random population contains a variety of different random constants. Once generated and inserted into the S-expressions of the population during the initial random generation, these constants remain constant.

After the initial random generation, the numerous different random constants arising from the ephemeral “R” atoms will then be moved around from tree to tree by the crossover operation. These random constants will become embedded in various sub-trees that then carry out various arithmetic operations on them. This "moving around" of the random constants is not at all haphazard, but, instead, is driven by the overall goal of achieving ever better levels of fitness. For example, a symbolic expression that is a reasonably good fit to a target function may become a better fit if a particular constant is, for example, decreased slightly. A slight decrease can be achieved in several different ways. For example, there may be a multiplication by 0.90, a division by 1.10, a subtraction of 0.8, or an addition of -0.004. Notice that if a decrease of precisely 0.09 in a particular value would produce a perfect fit, a decrease of 0.08 is more fit than a decrease of only 0.07. Thus, the relentless pressure of the fitness function in the natural selection process determines both the direction and magnitude of the numerical adjustments.

The process of combining random constants to achieve a desired goal is not always so direct as the simple example above where a decrease of 0.9 was desired. In one particular problem (discussed below in section 4.4.5) $\pi/2$ (about 1.57) was needed to solve the problem. In that problem, $\pi/2$ was found by first finding $2-\pi/2$ (about 0.429). The value $2-\pi/2$ was found by a succession of decreasing approximations in 11 steps. Starting with the available constant 1 and the available function SIN, (SIN 1) was first computed (0.841). Then the SIN of 0.841 was taken to obtain a still smaller number, namely 0.746. This result was then squared to obtain a still smaller number, namely 0.556. Then the SIN function was successively applied six more times to obtain a succession of still smaller numbers. The last one was 0.433. That is, the composition `(SIN (SIN (SIN (SIN (SIN (SIN (* (SIN (SIN 1)) (SIN (SIN 1)))))))))))` was used to compute the constant 0.433. Each successive step in this 11 step process produced a constant closer to what was needed. Each successive S-expression had slightly improved fitness than its predecessor.

To illustrate symbolic regression with constant creation, suppose we are given a sampling of the numerical values from the unknown curve $2.718x^2+3.1416x$ over 20 points in some domain, such as the interval [-1, +1]. In one run, the best individual S-expression in the population in generation 41 was

```
(+ (- (+ (* -.50677 X) (+ (* -.50677 X) (* -.76526 X))))
   (* (+ .11737) (+ (- X (* -.76526 X)) X))).
```

This S-expression is equivalent to $2.76X^2 + 3.15X$.

Symbolic regression has been similarly successfully performed on a large number of target expressions, including expressions such as $\mathrm{SIN}\ X + \mathrm{COS}\ X + X^2 + X$.

4.3.3. EMPIRICAL DISCOVERY

An important problem area in virtually every area of science is finding the empirical relationship underlying observed values of the variables measuring a system (Langley and Zytkow 1989). In practice, the observed data may be noisy and there may be no known way to express the relationships involved in a precise way.

4.3.3.1. ECONOMETRIC TIME SERIES

The problem of discovering such empirical relationships from actual observed data is illustrated by the well-known econometric “exchange equation” M=PQ/V, which relates the money supply M, price level P, gross national product Q, and the velocity of money V of an economy (Koza 1990). Suppose that our goal is to find the relationship between quarterly values of the money supply M2 and the three other elements of the equation.

In particular, suppose we are given the 112 quarterly values (from 1961:1 to 1988:4) of four econometric time series. The first time series is the annual rate for the United States Gross National Product in billions of 1982 dollars. This series is conventionally called “GNP82” by economists and we use the conventional notation of economists for such time series data herein. The second time series is the Gross National Product Deflator normalized to 1.0 for 1982 (called “GD”). The third series is the monthly interest rate yields of 3-month Treasury bills, averaged for each quarter (called “FYGM3”). The fourth series is the monthly values of the seasonally adjusted money stock M2 in billions of dollars, averaged for each quarter (called “M2”). The time series used here were obtained from the CITIBASE data base of machine-readable econometric time series (Citibank 1989). The CITIBASE data was accessed by an Apple Macintosh II computer using software provided by VAR Econometrics Inc. (Doan 1989).

The actual long-term historic postwar value of the M2 velocity of money in the United States is 1.6527 (Hallman et. al. 1989, Humphrey 1989) so that the “correct” solution is the multiplicative (non-linear) relationship

$$M2 = \frac{(GD * GNP82)}{1.6527}$$

However, we are not told a priori whether the functional relationship between the given observed data (the three independent variables) and the target function (the dependent variable M2) is linear, multiplicative, polynomial, exponential, logarithmic, or otherwise.

The set of available functions for this problem is F = {+, -, *, %, EXP, RLOG}.

The set of available terminals for this problem is T = {GNP82, GD, FYGM3, R}, where "R" is the ephemeral random constant atom allowing various random floating point constants to be inserted at random as constant atoms amongst the initial random LISP S-expressions. The "terminals" GNP82, GD, and FYGM3 provide access to the values of the time series for the 112 quarters. In effect, these "terminals" are functions of the unstated, implicit time variable.

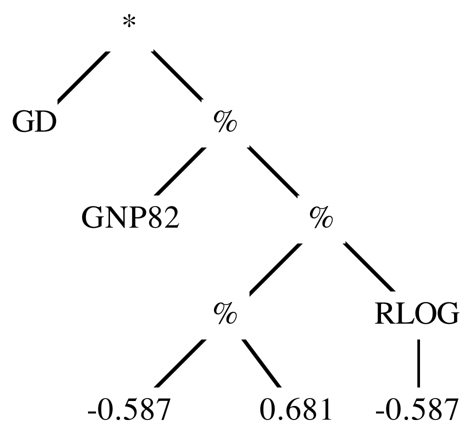
Notice that we are not told that the addition, subtraction, exponential, and logarithmic functions and the time series for the 3-month Treasury bill yields (FYGM3) are irrelevant to the problem.

The initial random population was, predictably, highly unfit. In one run, none of the individual S-expressions in the initial random population came within 3% of any of the 112 environmental data points in the time series for M2. The sum of errors between that best S-expression and the actual time series was very large (88448). Similarly, the best individuals in generations 1 through 4 came close to the actual time series in only a small number of cases (i.e. 7, 2, 3, and 5 of the 112 cases, respectively) and also had large sum of error measures (72342, 70298, 26537, 23627). However, by generation 6, the best individual came close to the actual time series in 41 of the 112 environmental cases and had an sum of errors of 6528.

In generation 9, the following S-expression for M2 emerged:

```
(* GD (% GNP82 (% (% -.587 0.681) (RLOG -0.587)))).
```

This individual is graphically depicted below:



Note that this S-expression for M2 is equivalent to

```
(% (* GD GNP82) 1.618),
```

or, more familiarly,

$$M2 = \frac{GD * GNP82}{1.618}$$

The S-expression discovered in the 9th generation comes within 3% of the actual values of M2 for 82 of the 112 points in the 28-year time series. The sum of the absolute errors between the S-expression discovered and the 112 points of the 28-year time series is 3765.2. The S-expression discovered here compares favorably to the “correct” “exchange equation” M=PQ/V (with a value of V of 1.6527) which had a sum of errors of 3920.7 and which came within 3% of the actual time series data for only 73 of 112 points in the 28-year time period studied.

This process was then repeated for two additional samples of the time period consisting of only the first and last two thirds of the available data points. In each case, the S-expression that emerged had the same form as above, but with a denominator consisting of a constant near, but not precisely equal to, the constant above. The performance of the S-expression discovered on the "in sample" points were then compared to the remaining "out of sample" data points and, in each case, a close fit was observed.

4.3.3.2. KEPLER’S THIRD LAW

Langley et. al. (1987) discuss the discovery of various scientific laws from empirical data. One example is the discovery in 1618 of Kepler’s third law of planetary motion, which states that the cube of a planet’s distance from the sun is proportional to the square of its period. That is,

$$\frac{D^3}{p^2} = c$$

In rediscovering Kepler’s Third Law using the genetic programming paradigm, we used the function set F = {+, -, *, %, SRT, SIN, COS} and the terminal set T = {DIST}. The environment consisted of 9 cases relating the distance DIST (in astronomical units) of each planet from the sun and the period P of the planet. The object was to find an S-expression for P in terms of DIST.

The most parsimonious versions of the solutions were S-expressions such as `(SRT (* DIST (* DIST DIST)))` and `(* DIST (SRT DIST))`. Less parsimonious correct solutions included S-expressions such as

```
(* DIST (+ (- DIST DIST) (+ (- DIST DIST) (SRT DIST)))) and
(- (* DIST (SRT DIST)) (SIN 0.0)).
```

Interestingly, the S-expression `(* DIST DIST)` appeared several times as a partially correct, ancestor of the correct final solution on several runs. Ten years before publishing the correct version of his third law, Kepler published this incorrect version.

4.3.3.3. CHAOS

One aspect of the study of chaos involves finding the function that fits a given sample of data. Often the functions of interest are recursive in the sense that the sequence of values of the function over time depends on one or more initial condition values.

One simple example of such a function is the function $x(t) = 4\ x(t-1)\ (1- x(t-1))$ over the unit interval [0,1] with $x(0) = 0.26$. For each time step, after the initial time t=0, the value of the function is computed using the value of the function at the previous time step. The environment consisted of the value of the function for times 0, 1, 2, ..., 50. The value of the function for time step 0 was the initial condition 0.26. The object was to discover the function for time steps 1, 2, ..., 50. The terminal PREV allows access to the value of the function at the previous time step.

In one run, the best individual in the 6th generation was

```
(* PREV (* 2.0 (* 2.0 (- 1.0 PREV)))).
```

Note that we did not search here for the value 0.26 as the initial condition.

4.3.4. TRIGONOMETRIC IDENTITIES

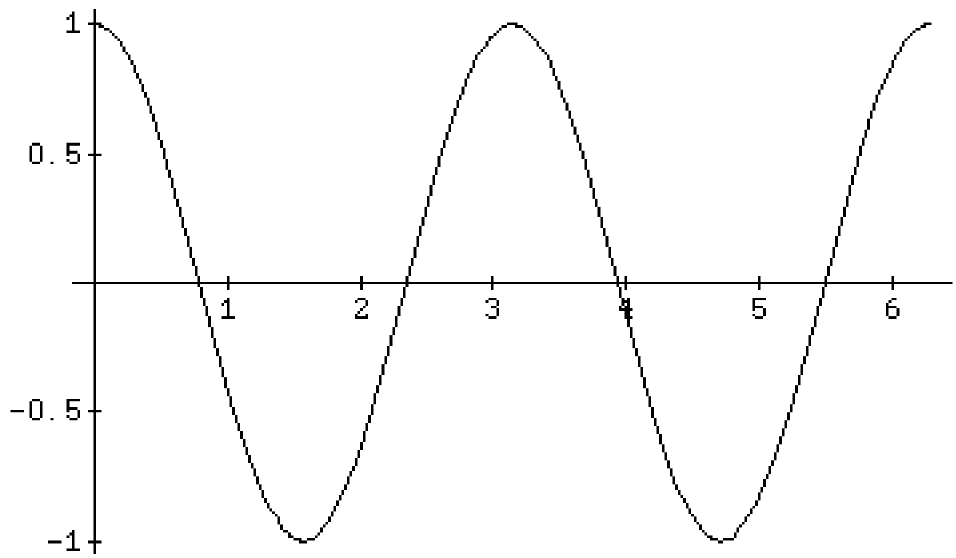
For this group of experiments, the problem was to derive various trigonometric identities. The environment consisted of 20 pairs of random Monte Carlo pairs of X and Y values between 0 and 2 π radians. In the first experiment, the problem was to find a LISP S-expression for Sin x+y. The available functions were SIN, COS, multiplication, addition, and subtraction. The expectation was that the algorithm would find `(+ (* (SIN X) (COS Y)) (* (SIN Y) (COS X)))`, or, more likely, a non-parsimonious, but equivalent, version of this S-expression. Interestingly, the initial result was `(SIN (+ X Y))`! That is, the algorithm rediscovered the original (but hidden) functional source of the original sequence. To avoid seeing this result (which was interesting and informative only once), the experiment was modified to find Cos 2X. The expectation was that the algorithm would find something like 2 Sin X Cos X. However, the algorithm instead produced the simpler identity Cos 2X = - Cos 2X.

Finally, the algorithm was given Cos 2X (which is equivalent to 1 - 2 Sin^2 X) and the addition and cosine function were intentionally deleted from the repertoire of available functions (to avoid merely getting an anticipated Cos X+X). The correct S-expression `(- (- 1 (* (SIN X) (SIN X))) (* (SIN X) (SIN X)))` was obtained after 13 generations in one run and the somewhat more parsimonious correct S-expression `(- 1 (* (* (SIN X) (SIN X)) 2))`, which is equivalent to 1 - 2Sin^2 x, was obtained after 16 generations in another run.

Another unexpected result arising from this experiment became the basis for our “constant creation” process described above. In one run where we were attempting to find a trigonometric identity for Cos 2X, the lengthy S-expression

```
(SIN (- (- 2 (* X 2))
     (SIN (SIN (SIN (SIN (SIN (SIN (* (SIN (SIN 1))
                                      (SIN (SIN 1)))))))))))
```

emerged in generation 30. This expression had near perfect (zero) raw fitness and scored 20 (out of 20) "near hits". Yet it is hardly obvious what this incomprehensible expression represented. By using the graphical features of the Mathematica™ software package (Wolfram 1988),we found that the graphical representation of this this lengthy expression was



That is, the S-expression was an extremely good fit to Cos 2X. It became clear that

```
2 - Sin[Sin[Sin[Sin[Sin[Sin[Sin[Sin[1]]^2]]]]]],
```

where the "1" is in radians, is approximately l.56721. Notice that π / 2 is about 1.57. Thus, the trigonometric identity Sin(π /2- 2X) for Cos 2X was discovered. More importantly, the algorithm created a needed constant (i.e. π /2) from the available ingredients (namely, the SIN function and the constant l) by repeatedly applying the Sin function to converge on the desired numerical result.

4.3.5. "DATA TO FUNCTION" SYMBOLIC INTEGRATION

Various approaches to the problem of "symbolic integration" are discussed in Mills (1987). In "data to function" symbolic integration, we are given numerical values for a sampling of points on an unknown curve and we desire to find the function, in symbolic form, that is the integral of the unknown curve.

In particular, suppose we are given a sampling of 50 numerical values from an unspecified curve. That is, we are given 50 pairs (x_i, y_i), where each $x_i < x_{i+1}$. The domain may be 0 to 2Π . The unspecified curve happens to be Cos x + 2x + 1. The goal is to find, in symbolic form, the a function which is a perfect fit or good fit to the integral of the unspecified curve from the given pairs of numerical points. That is, the goal is to find Sin x + x^2 + x in symbolic form from the 50 pairs (x_i, y_i).

For these purposes, we programmed a numerical integration routine to perform the numerical integration. It will be seen that the problem is, in fact, similar to the problem of symbolic regression discussed above, except for this additional numerical integration step. We first numerically integrate the curve (x_i, y_i) over the domain starting at x_1 and running to x_i so as to obtain a value for the integral of the unspecified function for each point x_i. Then, as each individual candidate function f_j is generated, we evaluate $f_j(x_i)$ so as to obtain 50 pairs $(x_i, f_j(x_i))$.

The raw fitness of an individual candidate function is the sum of the absolute values of differences between the individual candidate function at domain point x_i and the integral of the unspecified function up to domain point x_i.

As before, the desired solution can be viewed as a function from a hyperspace of functions that can be composed from the available functions (which are the same as above).

As before, the set of terminals contains just the single variable value x. Thus, the combined set of functions and terminals for this problem is C = {X, +, -, *, SIN, COS, RLOG} having 0, 2, 2, 2, 1, 1, and 1 arguments, respectively.

After 4 generations, the S-expression

```
(+ (+ (- (SIN X) (- X X)) X) (* X X))
```

emerged. This function has a very small error with respect to the 50 numerical points and it has a perfect score of 50 " near hits" in the sense that this function is within the criterion (0.01) of the integral of the unspecified curve for each of the 50 x_i values. This S-expression is equivalent to Sin $x + x^2 + x$.

To summarize, we found the functional form Sin $x + x^2 + x$ from the 50 given data points (x_i, y_i).

In another experiment, $x^4 + x^3 + x^2 + x$ was obtained as the symbolic integral of $4x^3 + 3x^2 + 2x + 1$.

4.3.6. "DATA TO FUNCTION" SYMBOLIC DIFFERENTIATION

In "data to function" symbolic differentiation, we are given numerical values for a sampling of points on an unknown curve and we desire to find the function, in symbolic form, that is the derivative of the unknown curve.

In particular, suppose we are given a sampling of 200 numerical values from an unspecified curve in the domain 0 to $\pi/2$. That is, we are given 200 pairs (x_i, y_i). The unspecified curve happens to be Sin $x + x^2 + x$. The goal is to find, in symbolic form, the a function that is a perfect fit or good fit to the derivative of the unspecified curve from the given pairs of numerical points. That is, the goal is to find Cos $x + 2x + 1$ in symbolic form.

For these purposes, we programmed a numerical differentiation routine to perform the numerical differentiation. It will be seen that the problem is, in fact, similar to the problems of symbolic regression and symbolic integration discussed above except that numerical differentiation is performed. In numerically differentiating the curve (x_i, y_i) for points other than endpoints of the domain, the derivative is the average of the slope of the curve between point x_{i-1} and x_i and the slope of the curve between point x_i and x_{i+1}. For the two endpoints of the domain, the slope is the (unaveraged) slope of the curve to the nearest point. If numerical differentiation is being performed, it is desirable to have a larger number of points (e.g. 200) because of the inherent relative inaccuracy of numerical differentiation as compared to numerical integration.

After 30 generations, the S-expression `(+ (+ (COS (- X X)) (+ X X)) (COS X))` emerged. This function has a very small error with respect to the 200 numerical data points and it has a perfect score of 200 "hits" in the sense that this function is within the criterion (0.01) of the y_i value for each of the 200 x_i values. This S-expression is equivalent to Cos $x + 2x + 1$.

In another experiment, $4x^3 + 3x^2 + 2x + 1$ was obtained as the symbolic derivative of $x^4 + x^3 + x^2 + x$.

4.4. SOLVING EQUATIONS

The "genetic programming" paradigm can be used to solve equations whose solution is in the form of a function that satisfies the given equation.

4.4.1. DIFFERENTIAL EQUATIONS

Differential equations are typically approached using analytic methods or numerical approximation methods. However, the problem of solving differential equations may be viewed as search in a hyperspace of compositions of functions and arguments for a particular composition which satisfies the equation and its initial conditions.

Consider the simple differential equation

$$\frac{dy}{dx} + y \text{ Cos } x = 0$$

having an initial value of y of 1.0 for an initial value of x of 0.0. The goal is to find a function which satisfies the equation, namely, $e^{-\text{Sin } x}$.

We start by generating 200 random values of the independent variable x_i over some domain, such as the unit interval [0, 1]. We call this set of values the "x-values." As the j-th individual candidate function f_j in the population is generated by the genetic algorithm, we evaluate $f_j(x_i)$ so as to obtain 200 pairs $(x_i, f_j(x_i))$. We call this 200 by 2 array of numbers the "unknown curve" or "candidate-curve."

We then numerically differentiate the "curve" $(x_i, f_j(x_i))$ with respect to the independent variable x_i to obtain the value of the derivative curve $(x_i, f_j'(x_i))$ for all 200 points. We then take the cosine of the 200 pairs of random values of x_i to obtain the curve consisting of the 200 values $(x_i, \text{Cos } x_i)$. We then multiply these 200 values by $f_j(x_i)$ so as to obtain 200 pairs $(x_i, f_j(x_i)^*\text{Cos } x_i)$. We then perform the addition $f_j'(x_i) + f_j(x_i)^*\text{Cos } x_i$ for all 200 points. To the extent that $f_j'(x_i) + f_j(x_i)^*\text{Cos } x_i$ is close to zero for the 200 values of x_i, the unknown (candidate) function f_j is a good approximation to the solution to the differential equation.

To facilitate the input of the differential equations into the system, a toolkit has been written to allow a differential equation to be written out in terms of the independent variable x (called x-values), the unknown function being sought (called "candidate-function"), differentiation (with respect to a specified variable), and any other function. These functions include addition, subtraction, multiplication, division, sine, cosine, etc. We apply a function to a "curve" by using the special "$" function and the name of the desired function. We adopt the convention that the right hand side of the equation is always zero. Thus, the left hand side of the differential equation involves the unknown (candidate) function. The "$" function can be applied to scalar constants as well as "curves." As an example, we would rewrite the left hand side of the above differential equation as

```
($'+ (differentiate candidate-function x-values)
     ($* candidate-function ($'cos x-values))).
```

This is interpreted as follows: The cosine function is applied to the independent variable (the x-values). Then, the result is multiplied by the unknown function y (the candidate function) to obtain an intermediate result. Then, the unknown function (candidate-function) is differentiated with respect to the independent variable (x-values) and the result added to the previously obtained intermediate result.

The sum of the absolute values of the differences between the left hand side of the equation and the right hand side of the equation (i.e. the zero function) is then computed. The closer this sum of differences is to zero, the better.

The fitness of a particular individual candidate (unknown) function is expressed in terms of two factors. The sum of absolute differences mentioned above represents 75% of the raw fitness of the function. The other 25% of the raw fitness is derived from the closeness of the candidate function to the initial condition, namely, the absolute value of the difference between the value computed by the candidate (unknown) function f_j for the initial condition and the given value for the initial condition.

The combined set of functions and terminals for this problem is C = {+, -, *, SIN, COS, RLOG, EXP, X} having 2, 2, 2, 1, 1, 1, 1, and 0 arguments, respectively.

In one run, the best individual in the initial random population (generation 0) was the function

$$e^{1-e^{x}}$$

Its raw fitness was 58.09 and only 3 of the 200 points were "near hits."

By generation 2, the best individual in the population was

$$e^{1-e^{\text{Sin } x}}$$

It raw fitness was 44.23 and only 6 of the 600 points were "near hits."

By generation 6, the best individual in the population was equivalent to

$$e^{-\text{Sin } x}.$$

The raw fitness head dramatically improved (decreased) to only 0.057. Moreover, 199 of the 200 points were now "near hits." This function is, in fact, the exact solution to the differential equation.

A second example of a differential equation is

$$\frac{dy}{dx} - 2y + 4x = 0$$

with initial condition such that y = 4 when x = 1.

In one run, the best individual in the 28th generation was

(+ (* (EXP (- X 1)) (EXP (- X 1))) (+ (+ X X) 1)).

This is equivalent to

$$e^{-2}e^{2x} + 2x + 1,$$

which is the exact solution to the differential equation.

A third example of a differential equation

$$\frac{dy}{dx} = \frac{2 + \text{Sin } X}{3(y-1)^2}$$

with initial condition such that y = 2 when x = 0.

In one run, the best individual in the l3th generation was

```
(- (CUBRT (CUBRT 1))
   (CUBRT (- (- (- (COS X) (+ 1 (CUBRT 1))) X) x))),
```

where CUBRT is the cube root function. This is equivalent to

$$1 + (2 + 2x - \text{Cos } x)^{1/3},$$

which is the exact solution to the differential equation.

Note that when the initial condition of the differential equation involves only a value of the function itself (as is typically the case when the differential equation involves only first derivatives), any point in the domain of the independent variable (X) may be used for the initial condition. On the other hand, when the initial condition of the differential equation involves a value of a derivative of the function (as may be the case when the differential equation involves second derivatives or higher derivatives), it is necessary that the value of the independent variable (X) involved in the initial condition be one of the points in the random set of points x_i (and preferably an internal point of the domain). This is necessary to allow the first derivative (or higher derivative) of the unknown (candidate) function to be evaluated for the initial condition point.

4.4.1.1. POWER SERIES SOLUTION TO A DIFFERENTIAL EQUATION

It is also possible to discover the power series program for e^x when an exponential function is part of the solution to a differential equation.

In this experiment, we demonstrate the use of two tools which are commonly used in computer programming, namely iteration and the ability to give a name to the results of a calculation (or the result of a sub-program) so that it can subsequently referred to and subsequently used.

For this experiment, the problem is to find the solution to

$$\frac{dy}{dx} - y = 0$$

having an initial value of y of 2.718 for an initial value of x of 1.0. In effect, the problem is to compute e^x using the power series

$$\sum_{j=0}^{\infty} \frac{x^j}{j!}$$

The functions available for this problem include addition (+), multiplication (*), and the modified division operation % (which returns a value of zero when division by zero is attempted), the "set" operator SA, and an iterative summation operator SIGMA. The terminals available for this problem include the variable X, an iterative summation index II, and the settable variable AAA.

The "set" operator SA has one argument and sets the global variable AAA equal to the value of its argument. The set function allows a computer program to assign a name to the results of a calculation (or the result of a sub-program) so that it can subsequently refer to and subsequently use that result.

In writing computer programs, computer programmers often mistakenly use a variable that has not yet been defined by their program. Depending on the programming language or machine involved, such undefined variables typically either cause the computer program to halt or they are assigned a default value (which may well be inappropriate to the specific problem involved). Our purposes here are best served by not having the evaluation of any one individual halt the overall operation here. Thus, we assign a default value to any undefined variable. Since this problem involves real-valued variables, the default value here should be a floating point number. In order to simplify the particular problem here, we have made the default value 1.0 for undefined variables. Note that a given S-expression typically changes the value of a settable variable during the course of the program (and may do so many times).

The iterative summation operator SIGMA has one argument, called WORK, and performs a function similar to the familiar summation operator Σ in mathematics. In particular, the operator SIGMA evaluates its WORK argument repeatedly until a summand is encountered that is very small (e.g. less than .000001 in absolute value). The operator SIGMA then returns the value of its accumulated sum.

The operator SIGMA is similar to the previously described iterative DU ("Do Until") operator in that an indexing variable II is available inside the SIGMA operator as it iterates. While a SIGMA operator is performing iterations, an iteration variable II counts the number of iterations (starting with l). This variable can be one of the terminals in the set of available terminals. Thus, if the argument WORK happens to contain II, the SIGMA operator becomes a summation over the indexing variable. Of course, if it does not, the SIGMA operator merely accumulates a sum of summands that are independent of II (but which may, nonetheless, change due to the operation of settable variables or other side effects).

Since individual S-expressions in the population are not generally or necessarily very small in absolute value, there is no guarantee that the operator SIGMA will terminate. Therefore, it is a practical necessity (when working on a serial computer) to place limits on both the number of iterations allowed by any one execution of a SIGMA operator and to place a similar limit on the total number of iterations allowed for all SIGMA operators that may be evaluated in the process of executing any one individual S-expression for any particular environmental case. Note that even when a SIGMA operator times out, it nevertheless returns the sum accumulated up to that time.

The LISP S-expression `(SIGMA (SA (* AAA (% X II))))` is a parsimonious LISP S-expression for computing the value of the power series for e^X - 1 for a given value of X. This S-expression consists of a SIGMA operator that starts by setting AAA to the result of multiplying the value of AAA (which initially is l) by X and dividing by the iteration variable II. As this iterative process continues, the summands successively consist of the powers of X divided by the factorial of the iteration number. When the current settable variable AAA gets very near zero, the SIGMA operator terminates and returns its accumulated value (namely, the last overall sum). Note that if the termination predicate is ill-formed (as it often is), the iterative summation operator will "time out" when the limit on the number of iterative steps is reached (e.g. 15).

In one run, we obtained `(SIGMA (SA (* SA AAA) (SA (% X II)))))` as the best individual LISP expression on the 13th generation. When simplified, this LISP S-expression is equivalent to the correct solution to the differential equation and its initial conditions.

4.4.2. INTEGRAL EQUATIONS

Integral equations are equations that involve the integral of the unknown function. Integral equations can be solved with the same general approach as above, except for the additional step of taking the integral of the candidate function.

An example of an integral equation is

$$y(t) - 1 + 2 \int_{r=0}^{r=t} \text{Cos}(t-r)\, y(r)\, dr = 0$$

In one run, we found the solution to this integral equation, namely,

$$y(t) = 1 - 2te^{-t}$$

Note that the process of integration creates a variable (r, in this case), which is similar to the indexing variable II of an iterative loop (described above in connection with the "DU," "DUL," and "SIGMA" functions).

4.4.3. INVERSE PROBLEMS

Suppose we have a set of data consisting of various (x_i, y_i) pairs such as (9,6), (16,8), (25,10), (36,12), (2.25, 3.0), etc.

Symbolic regression would reveal that the dependent variable y_i is twice the square root of the independent variable x_i. That is,

$$y_i = 2 \sqrt{x_i}.$$

The problem of finding the inverse function starts with a set of (x_i, y_i) pairs of data such as (6,9), (8,16), (10,25), (12,36), (3.0, 2.25), etc. The conclusion is that the dependent variable y_i is the square of half of the independent variable x_i. That is,

$$y_i = \frac{x_i^2}{2}$$

It will be seen that the problem of finding an inverse function for a given set of data is similar to the problem of symbolic regression discussed above, except for the additional step of switching the roles of the independent and dependent variables of the data set.

4.4.4. SOLVING FUNCTIONAL EQUATIONS

Consider the following functional equation:

$$f(2x) - 1 + 2\mathrm{Sin}^2 x = 0.$$

The goal is to solve this equation for the function f, which, when substituted into the equation, satisfies the equation.

As before, we begin by selecting a set of random points in a suitable domain. In particular, we select 50 points x_i in the domain of real numbers between -3.14 and +3.14. We store these 50 values in a vector. We then compute another vector of 50 values corresponding to the sine of each x_i. We then compute another vector of 50 values corresponding to the square of the sine of each x_i. We then compute another vector corresponding to twice the square of the sine of each x_i. Each of these computed vectors can also be viewed as a curve since we can think of the points for $2\mathrm{Sin}^2 x$ being plotted graphically on axes. Similarly, we set up a vector constant of 50 occurrences of the constant 1 (the "constant curve"). We then subtract this "constant curve" from the "curve" just computed for $2\mathrm{Sin}^2 x$. Finally,we consider each of the S-expressions f_j in the current populations of individuals. Since the argument for the unknown function is 2x (instead of just x), we must first perform the step of multiplying the 50 x_i values by 2. We then compute the curve for f(2x) using the S-expression f_j.

If we happen to have the function f that exactly satisfies the equation, the new "curve" computed will consist of all zeroes. In any case, the value of the left hand side $f(2x) - 1 + 2\mathrm{Sin}^2 x$ corresponds to the fitness of the function.

In one run, the S-expression below emerged on generation 7 with a raw fitness of zero:

```
(* 1 (COS (+ X X))).
```

This S-expression is equivalent of Cos 2x and solve the functional equation. That is, when Cos 2x is substituted into the equation

$$f(2x) - 1 + 2\mathrm{Sin}^2 x = 0,$$

the equation is satisfied (i.e. the left hand side evaluates to zero for each random x_i).

4.4.5. SOLVING GENERAL MATHEMATICAL EQUATIONS FOR NUMERIC VALUES

An important special case of the process of solving functional equations occurs when the set of arguments (atoms) consists only of numeric constants. That is, there are no variable arguments (such as X) in the set of arguments used to construct the S-expressions. In this special case, the process can be used to solve a general mathematical equation for its numerical roots.

For example, consider the simple equation, with two identical roots of $\sqrt{2}$, which one would conventionally write as

$$x^2 - \sqrt{2}\ x + 2 = 0.$$

For our purposes here, we rewrite this equation as the functional equation

$$f^2(x) - \sqrt{2}\, f(x) + 2 = 0,$$

where the function f(x) is the unknown (instead of the variable X being the unknown).

We proceed by using a set of functions that contains functions such as addition, subtraction, multiplication, and division. The set of terminals (arguments), however, consists only of the ephemeral random constant atom ("R"). Note that X does not appear in this set of terminals (arguments). As a result, the set of S-expressions contains only random compositions of random constants. Typical S-expressions might be

```
(+ 0.234 (* -0.685 0.478)) and
(* (* 0.537 -1.234) (+ 1.467 0.899)).
```

As before, 50 random values of x_i are selected in a suitable domains (such as -2.0 to +2.0). A "curve" is then built up by squaring each x_i. Next, each x_i is multiplied by $\sqrt{2}$ and this result is subtracted from the square of each x_i . The constant value 2.0 is then added to each of the 50 values. The next step is to evaluate the fitness of each of the 300 individual S-expressions f_j in the population. Each S-expression in this problem has a particular numeric value because the initial population of S-expressions contained only constants. Its value does not depend on x_i. Thus, when each f_j is evaluated for fitness, the value is the same for all 50 cases (because the value f_j does not depend on x_i). As before, the sum of these 50 (identical) values is the fitness of the S-expression f_j. If the S-expression causes the left hand side of the equation (i.e. the raw fitness side) to be zero, that S-expression (which is, in fact, a numeric constant value) satisfies the equation.

In one run, the best individual S-expression in the 42nd generation evaluated to 1.41417. This is within 0.00004 of the value of $\sqrt{2}$, which is approximately 1.41421.

Note that this genetic approach to solving equations for numeric values produces quite precise values. This result is contrary to the conventional view that genetic algorithms are only good for searching for the general neighborhood of a correct answer in a large search space.

This view is perhaps correct when applied to conventional genetic algorithms operating on character strings whose length is fixed in advance. However, where the size and shape of the solution is allowed to dynamically vary as the problem is being solved, it is possible to search a large search space for the correct general neighborhood of the solution and then, by adaptively changing the representation scheme, converge closely onto the precise correct value.

4.5. CONCEPT FORMATION

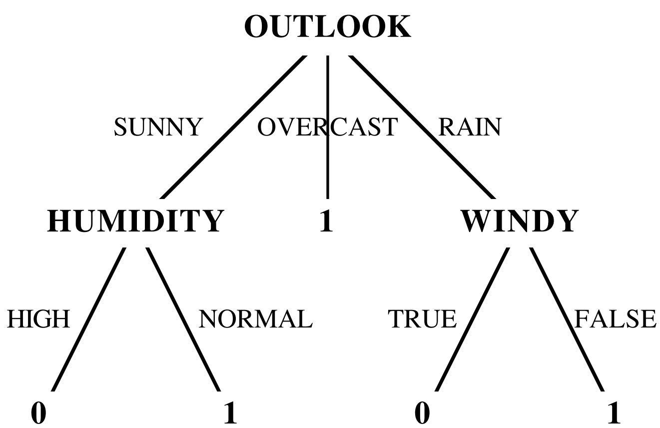
Quinlan (1986) initiated development of a particularly effective family of hierarchical classification systems for inducing a decision tree from a limited number of training case examples. In ID3 (and various other systems of the ID3 family), the goal is to partition a universe of objects into classes. Each object in the universe is described in terms of various attributes. The system is first presented with a set of training case examples which consist of the attributes of a particular object and the class to which it belongs. The system then generates a decision tree which hopefully can then be used to classify a new object correctly into a class using the attributes of the new object. The external points (leaves) of the decision tree are the eventual class names. The internal points of the decision tree are attribute-based tests which have one branch emanating from the decision point for each possible outcome of the test.

The induction of such decision trees for classifying objects can be approached by genetically breeding LISP S-expressions for performing this task. In particular, the set of terminals is the set of class names.

The set of functions is the set of attribute-based tests. Note that this set of attribute-based tests are always assumed to be given and available for solving induction problems via decision trees of the

ID3 family. Notice that ID3 is similar to the genetic programming paradigm in that the set of functions is given. Each function has as many arguments as there are possible outcomes of that particular test. When a particular object is presented to the LISP S-expression (i.e. the decision tree), each function in the S-expression tests one attribute of the object and returns the particular one of its arguments designated by the outcome of the test. If the designated argument is a terminal, the function returns the class name. When the S-expression is fully evaluated in LISP's usual left-oriented depth-first way, the S-expression as a whole thus returns a class name. That is, the S-expression is a decision tree that classifies the new object into one of the classes.

To demonstrate the technique of genetically inducing a decision tree, we apply this approach to the small training set of 14 objects presented in Quinlan (1986). In Quinlan's problem, each object has four attributes and belongs to one of two classes ("positive" or "negative"). The attribute of "temperature", for example, can assume the possible values hot, mild, or cool. Humidity can assume the values of high or normal. Outlook can assume values of sunny, overcast, or rain. Windy can assume values of true or false. The decision tree presented by Quinlan as the solution for this problem is shown below:



If, for example, the OUTLOOK of a particular object is sunny and the HUMIDITY is high, then that object is classified into class 0 (negative).

In order to genetically induce the decision tree, each of the four attributes in this problem is converted into a function. For example, the function "temperature" operates in such a way that, if the current object has a temperature of "mild," the function returns its second argument as its return value. The other attributes in this problem, namely "humidity", "outlook", and "windy", are similarly converted to functions. The function set for this problem is therefore F = {TEMP, HUM, OUT, WIND} with 3, 2, 3, and 2 arguments, respectively. The set of terminals for this problem is T = {0, 1} since there are two classes.

In one run, the LISP S-expression

```
(OUT (WIND 1 0) (WIND 1 1) (HUM 0 1)
```

emerged on the 8th generation with a maximal fitness value of 14 (i.e. it correctly classified all 14 training cases). Since (WIND 1 1) is equivalent to just the constant atom 1, this S-expression is equivalent to the decision tree presented in Quinlan (1986) using ID3.

4.6. AUTOMATIC PROGRAMMING

The problem of automatic programming requires developing a computer program that can produce a desired output for a given set of inputs. Early provocative work in this area includes Friedberg (1958 and 1959).

4.6.1. SOLVING PAIRS OF LINEAR EQUATIONS

For this experiment, the problem is to find the computational procedure for solving a pair of consistent non-indeterminate linear equations, namely $a_{11}x_1 + a_{12}x_2 = b_1$ and $a_{21}x_1 + a_{22}x_2 = b_2$ for the first of its two real-valued variables (x_1). The environment consisted of a suite of 10 pairs of equations. Using a suite of equations minimizes the chance of being misled. Without loss of generality, the coefficients of the equations were prenormalized so the determinant is 1. The set of available functions is F = {+, -, *} and the set of available terminals is T = {A11, A12, A21, A22, B1, B2}.

The worst individual from the initial random population (i.e. generation 0) has a raw fitness value of 119051.

The average raw fitness for generation 0 (the initial random generation) is 2622. This value serves as a baseline by which to measure future (non-random) performance.

The average raw fitness of the population immediately begins improving from the baseline value for generation 0 of 2622 to 632, 341, 342, 309, etc. In addition, the worst individual in the population also begins improving from 119051 for generation 0 to 68129, 2094, etc.

The best individual from generation 0 is `(+ (- A12 (* A12 B2)) (+ (* A12 B1) B2))` and has a raw fitness value of 125.8. The best individual begins improving and has a value of 106 for generations 1 and 2, 103 for generation 3 through 5, 102 for generations 6 through 16, and 102 for generations 17-20. The computational procedure `(+ (- A12 (* A12 B2)) (* A22 B1))` that appeared in generations 21 and 22 had a fitness value of 62 and differed from the known correct solution only by one additive term -A12. The best individual for generations 23 through 26 is a similarly close S-expression `(+ (- A22 (* A12 B2)) (* A22 B1))` with a raw fitness value of 58. Starting with generation 27, a perfect solution for x_1 emerges, namely `(- (* A22 B1) (* A12 B2))`.

The average normalized fitness for each generation shows a gradual, generally upwards trend from generation to generation starting with .0017 at generation 0, improving to .0029 at generation 1, and reaching .0059 for generation 26. Between generation 27 and generation 30, the average normalized fitness rises sharply to .39 as the known perfect solution begins to dominate the population.

4.6.2. QUADRATIC EQUATIONS

For this experiment, the problem is to solve the quadratic equation $x^2 + bx + c = 0$ for a complex-valued root. The available functions were multiplication, subtraction, a modified division operation %, and the Common LISP complex-valued square root function SQRT (denoted by √ here). The modified division function % returns 0 when division by zero is attempted. The square root function √ returns a LISP complex number. Thus, for example, (√ -4) calls for the square root of -4 and would evaluate to the LISP complex number `#C(0, 2)`. The set of terminals is T = {B, C}.

The environment consisted of a suite of 10 quadratic equations (with some purely real roots, some purely imaginary roots, and some complex-valued roots). For each of the l0 equations in the environment, a given individual S-expression was evaluated to obtain a (generally) complex number. The square root of the square of the real part of the complex number produced by the individual LISP S-expression and the square of the imaginary part of the complex number produced by the individual LISP S-expression was computed. These distance values were then summed over the l0 quadratic equations in the environmental test suite to obtain the fitness value of the given S-expression.

In one run, a correct solution to the problem emerged at generation 22, namely, the S-expression `(- (√ (- (* (% B 2) (% B 2)) C)) (% B 2))`, which is equivalent to the well-known solution.

4.6.3. OTHER COMPUTATIONAL PROBLEMS - FINDING PRIMES

A wide variety of other computational problems can be viewed as requiring the generation of a computer program to solve the problem.

One such example is finding the prime numbers. We will use this problem to illustrate a third iterative control structure, namely, an operator equivalent to the “FOR” loop found in many programming languages. A prime number is a positive integer which is evenly divisible only by itself and one. The problem of finding primes can be viewed as finding a function over the positive integers that returns the number of divisors. If the number of such divisors is greater than two, then the number is not a prime (i.e. is a composite number). If the number of such divisors is two (or perhaps one, depending on how one handles the argument one), then the number is a prime. If the test suite is the first 64 integers, then the space of possible two-valued functions for this problem is of size 2^{64} (and therefore equals the search space for the Boolean 6-multiplexer).

Two approaches to this problem were used. In both approaches, the variable atom J is an integer between 1 and 64. The objective is to determine whether J is a prime. A settable variable CNC is available and it is initially zero. In both approaches, the function CZ is available. The function (CZ J M) adds one to CNC if J modulo M is zero.

In the first approach, an iterative operator DUL was used. The DUL operator is equivalent to the “FOR” loop found in many programming languages, whereas the DU operator is equivalent to the “REPEAT...UNTIL” loop and the SIGMA operator is equivalent to the Σ notation for infinite series in mathematics. The operator DUL ("Do-Until-Loop") has two arguments, namely, the work WORK and the number of iterations NLOOP to be performed. It is similar to the previously described iterative operator DU ("Do-Until") and the previously described iterative summation operator SIGMA in that an iteration variable II is available inside the DUL operator for possible incorporation into the WORK argument or the NLOOP argument and in that “time out” limits must be established for this operator.

The combined set of functions and terminals for this first approach is C = {DUL, CZ, J, II}. If an S-expression returned a value that was not greater than 2, it is deemed to be a prime. Otherwise, it is deemed to be a composite number. Fitness is the number of integers between l and 64 that were correctly classified. The S-expression `(DUL (CZ J II) J)` is a parsimonious and completely correct solution to this problem. This solution was obtained in several runs of the program.

In the second approach, the only operative function available was CZ. The passive function PROGN was included in the set of available functions to allow a sequence of functions to be performed. The available terminals were J, CNC, and the integers up to 8 (i.e. the square root of 64). An appropriate sequence of CZ functions with appropriate arguments can thus function as a sieve. Moreover, partially correct structures can easily develop. In one run, (CZ J 2) appeared as the best individual of generation 0. Then, `(PROGN (CZ J 2) (CZ J 7) CNC)` appeared with slightly better fitness. Then, `(PROGN (CZ J 3) 7 (PROGN (CZ J 2) (CZ J 7) CNC ))` appeared with even better fitness. Finally, `(PROGN (CZ (PROGN (CZ J 3) 5 J) 5) 7 (PROGN (CZ J 2) (CZ J 7) CNC))` appeared in generation 5. This S-expression is as close to a solution as is possible with the available terminals and functions.

4.7. PATTERN RECOGNITION

Hinton (1988) has discussed the problem of translation-invariant recognition of a one-dimensional shape in a linear binary retina (with wrap-around). In the simplified experiment here, the retina has 6 pixels (with wrap-around) and the shape consists of three consecutive binary 1's. Thus, 001110, 000111, and 100011 are among the 6 possible instances of the shape.

The functions available are a zero-sensing function H0, a one-sensing function H1, ordinary multiplication, and a disjunctive function U. The terminals available are the integers 0, l, and 2, and a universally quantified terminal k. The function H0 (or H1) takes two arguments and returns the integer 1 if there is a 0 (or 1) in the position equal to the sum of the two arguments (modulo the retina length) and returns the integer 0 otherwise. Thus, one argument of these functions can potentially serve as a positional pointer and the other can potentially serve as a displacement. The universally quantified terminal k assumes all integral values over the retinal length. The disjunctive function U takes two arguments and returns the integer 1 if either argument is non-zero and returns the integer 0 if both arguments are 0. The ordinary multiplication function * serves as a conjunctive function and returns the integer 1 if all arguments are non-zero and returns the integer 0 if any argument is 0.

The functions U and * so defined resolve potential type problems that would otherwise arise when integers identify positions in the retina. Although LISP is comparatively tolerant as to typing, pattern recognition problems seem to require the ability to freely combine numerical concepts such as positional location (either absolute or universally quantified) and relative displacement (e.g. the symbol 2 pixels to the right) with Boolean concepts (e.g. a particular disjunctive and conjunctive combination of features indicates a particular shape). One does not want to specify or restrict a priori the kind of combination of functions available to solve the problem.

Initial random individuals include contradictions such as `(* (H0 2 2) (H1 2 2))`, inefficiencies such as `(U (H0 2 1) (H0 1 2))`, irrelevancies such as `(U (H0 0 0) (H1 2 0))`, and nonsense such as `(U 2 (* k (H1 0 0)))`. In one particular run, the number of mismatches for the best individual of generation 0 was 48 and rapidly improved to 40 for generations 1 and 3. It then improved to 0 mismatches in generation 3 for the individual `(* 1 (* (H1 K 1) (H1 K 0) (H1 K 2)) 1)`. Ignoring the extraneous outermost conjunction of two 1's, this individual returns a value of the integer 1 if and only if a binary 1 is found in the retina in positions 0, 1, and 2 (each displaced by the same constant k).

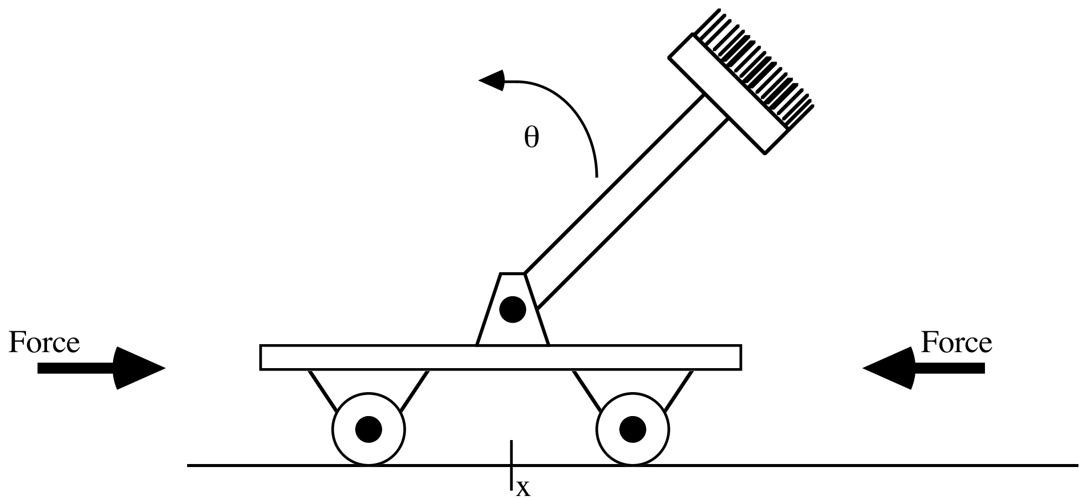
4.8. NON-LINEAR OPTIMAL CONTROL STRATEGY FOR BALANCING A BROOM

The problems of centering a cart and balancing a broom (inverted pendulum) by applying a "bang bang" force from either direction are well-known problems in control theory. The cart centering problem has been previously studied in the genetic algorithm field in connection with Holland classifier systems (Goldberg 1983). The broom balancing problem has been studied extensively in connection with neural networks (Widrow 1963, Barto et. al. 1983, Widrow 1987, Anderson 1988, Anderson 1989).

The genetic programming paradigm can be successfully used to solve the cart centering problem and two dimensional broom balancing problem (Koza and Keane 1990a) as well as the three dimensional broom balancing problem (Koza and Keane 1990b).

The three dimensional broom balancing problem involves a push cart with mass m_c moving on a one dimensional frictionless track and a broom (inverted pendulum) of mass m_b pivoting on the top of the cart. The broom has an angle θ (measured from the vertical) and an angular velocity ω. The distance from the center of mass of the broom to the pivot point on the cart is λ. There are four state variables of this system, namely, position x, velocity v, angle θ,and angular velocity ω.

A “bang bang” force of fixed magnitude F is applied to the center of mass of the cart from either the left or right direction at each time step τ**.** The constants here are m_c=0.9 kilogram, m_b=0.1 kilogram, gravity g=1.0 meters/sec^2, time step τ=0.02 seconds, and length λ =0.8106



meters.m

The state transitions of this cart and broom system are expressed by non-linear differential equations. At each discrete time step τ, the current state of the system and the force being applied at that time step are used to compute the state of the system at the next time step. In particular, the angular acceleration $\phi(t)$ at time t is given by (Anderson 1988) as

$$\Phi(t) = \frac{g \, Sin\,\theta + Cos\,\theta \, \dfrac{-F - m_p \lambda \, \omega \, \theta^2 \, Sin\,\theta}{m_c + m_p}}{\lambda \left[\dfrac{4}{3} - \dfrac{m_p \, Cos^2 \theta}{m_c + m_p}\right]}$$

The angular velocity $\omega(t+1)$ of the broom at time t+1 is therefore

$\omega(t+1) = \omega(t) + \tau \, \Phi(t)$ and the angle $\theta(t+1)$ at time t+1 is

$\theta(t+1) = \theta(t) + \tau \, \omega(t)$, using Euler approximate integration.

The acceleration a of the cart is given by

$$a(t) = \frac{F + m_p \lambda \, [\theta^2 \, Sin\,\theta - \omega \, Cos\,\theta]}{m_c + m_p}$$

The velocity at time t+1 is therefore $v(t+1) = v(t) + \tau \, a(t)$ and the position $x(t+1)$ at time t+1 is $x(t+1) = x(t) + \tau \, v(t)$.

The problem is to find a control strategy (i.e. LISP S-expression) that specifies how to apply the "bang bang" force so that, after starting in any random initial state, the cart is brought to rest and the broom becomes balanced <u>in minimal average time</u>.

The set of functions used for this problem consisted of addition (+), subtraction (-), multiplication (*), the sign function (SIG), the absolute value function (ABS), the square root of absolute value function (SRT), the square function (SQ), the cube function (CUB), and the greater-than function (GT). The greater-than function GT is a real-valued logic function of two arguments that returns +1 if the first argument is greater than the second argument and returns -1 otherwise. The use of real-valued logic allows arbitrary composition of functions in the function set.

The set of terminals for this problem consisted of velocity v, angle θ, angular velocity ω, and the ephemeral constant atom "R" for real-valued random constants.

Each control strategy is executed (evaluated) on every time step of each environmental case. When a particular control strategy (i.e. LISP S-expression) evaluates to any positive value for particular values of state variables at a particular time step, the force F is applied from the positive direction. Otherwise, the force is applied from the negative direction. For example, in the figure below, the current point in the v- θ - ω space is shown by the ball which is above the control space so that the force F is applied from the positive direction.

4.8.1 CASE I

The environment consists of 10 initial environmental starting condition cases. Position is chosen randomly between -0.2 and +0.2 meters. Velocity v is chosen randomly between -0.2 and +0.2 meters/second. Angle θ is chosen randomly between -0.2 radians (about 11.5 degrees) and +0.2 radians. Angular velocity ω is chosen randomly between -0.2 and +0.2 radians per second. The force F is 1.0 Newtons.

Time was discretized into 600 time steps of .02 seconds. The total time available before the system "times out" for a given control strategy is thus 6 seconds. If the square root of the sum of the squares of the velocity v, angle θ , and angular velocity ω (i.e. the "norm") is less than 0.07 (the "criterion"), the system is considered to have arrived at its target state (i. e. the broom balanced and the cart at rest). If a particular control strategy brings the system to the target state for a particular environmental starting condition case, its fitness for that environmental case is the time required (in seconds). If a control strategy fails to bring the system to the target state before it "times out", its fitness for that environmental case is set to 6 seconds. The "fitness" of a control strategy is the average time for the strategy over all 10 environmental cases.

The initial population of random control strategies in generation 0 includes many highly unfit control strategies, including totally blind strategies that ignore all the state variables, partially blind strategies that ignore some of the state variables, strategies that repetitively apply the force from only one direction, strategies that are correct only for a particular few specific environmental cases, strategies that are totally counter-productive, and strategies that cause wild oscillations and meaningless gyrations.

In one run, the average time consumed by the initial random strategies in generation 0 averaged 5.3 seconds (i.e. about 3 times the average time of 1.85 seconds consumed by the pseudo optimal strategy). In fact, a majority of these 300 random individuals "timed out" at 6 seconds (and very likely would have timed out even if more time had been available). However, even in this highly unfit initial random population, some control strategies are somewhat better than others.

The best single control strategy in generation 0 was the non-linear control strategy $v^2 + \theta$ which averaged 3.77 seconds. Note that this control strategy is partially blind in that it does not even consider the state variable ω in specifying how to apply the “bang bang” force.

The average population fitness improved to 5.27, 5.23, 5.15, 5.11, 5.04, and 4.97 seconds per environmental case in generations 1 through 6, respectively. In generation 6, the best single individual was the non-linear control strategy

$$\theta + \sqrt{(|\omega| - \omega^2)}.$$

This individual performed in an average of 2.66 seconds. Moreover, this individual succeeded in bringing in 7 out of the 10 environmental starting conditions cases to the target state. This compares to only 4 such "hits" for the best single individual of generation 0 (where, in fact, about two thirds of the individuals in the population scored only 1 "hit").

By generation 10, the average population fitness had improved further to 4.8 seconds and scored 8 hits. The best single individual was $\theta + 2\,\omega - v^2$. Note that this individual is not partially blind and considers all three state variables.

By generation 14, the average fitness had improved to 4.6 seconds. And, for the first time, the high point of the "hits histogram" moved from 1 (where it started at generation 0) to a higher number (namely 4). In generation 14, 96 of the 300 individuals scored 4 "hits". This undulating left-to-right "slinky-like" motion in the "hits histogram" occurs as the system progressively learns.

In generation 24, we attained one individual that scored of 10 hits. The best individual in generation 24 is the non-linear control strategy $v + \theta + 2\omega + \theta^3$. This individual had fitness 2.63 seconds. By generation 24, the population average fitness improved to 4.2 seconds.

Generation 27 is the last time when we see a linear control strategy as the best individual in the population. The best single individual in the population at generation 27 was

$v + 2\theta + 3\omega$. It scored 10 hits and had fitness of 2.16 seconds. Note that the computer program or control strategy can be viewed as defining the optimal control surface that separates the state space into parts. In this case, the control surface is merely a plane, whereas, for most generations, it is a non-linear surface.

In generation 33, the best single individual bears a resemblance to the ultimate solution we attain in generation 46. In generation 33, the best single individual is $8\ \omega\ 3 + v + \theta + \omega$.

This individual had fitness 1.57 seconds. Moreover, 45 individuals in the population of 300 in generation 33 scored 10 hits.

By generation 35, the high point of the "hits histogram" of the population moved from 4 hits to 10 hits. In particular, 120 of 300 individuals in the population scored 10 hits.

In generation 46, the best single individual in the population was the 8-term non-linear control strategy

$$v + 2\theta + \omega + 8\omega 3 + \omega 2 + v\omega + v\sqrt{}\theta + \omega\sqrt{}\theta .$$

After its discovery, this single best control strategy was retested on 1000 additional random environmental starting condition points. It performed in an average of 1.51 seconds. In another test, it averaged 2.65 seconds on the 8 corners of the cube. In another test, it took 4.24 seconds for the worst two corners of the cube (i.e. where the velocity, angle, and angular velocity have the same sign). It never timed out for any internal point or any corner point of the cube.

A benchmark pseudo optimum strategy averaged 1.85 seconds over the 1000 random environmental starting condition cases in the retest. It averaged 2.96 seconds for the 8 corners of the cube. Moreover, it was unable to handle the 2 worst corners of the cube.

These results (in seconds) are summarized in the table below:

CASE I - PERFORMANCE FOR 3-DIMENSIONAL BROOM BALANCING PROBLEM

Control Strategy	1000 Points	8 Corners	Worst 2 Corners
Benchmark Pseudo Optimum	1.85	2.96	Infinite
$v + 2\theta + \omega + 8\omega 3 + \omega 2 + v\omega + v\sqrt{\theta} + \omega\sqrt{\theta}$	1.51	2.65	4.24

There is no known solution for this problem nor is there any specific test we can perform on an apparent solution that we obtain to verify that it is the optimum. We do know that the control strategy

$$v + 2\theta + \omega + 8\omega 3 + \omega 2 + v\omega + v\sqrt{\theta} + \omega\sqrt{\theta}.$$

had the best time of those we discovered; that we discovered a number of other control strategies that were slightly worse (and hence this control strategy seemed to be the result of some kind of convergent process); and that it is slightly better than the benchmark psuedo optimum strategy.

Two histograms provide a graphical picture of the learning of the population from generation to generation. The “hits histogram” shows the number of individuals in the population that correctly handle a particular number of environmental cases. The “fitness histogram” shows the number of individuals in the population with whose fitness value lies in a particular decile of the fitness values.

The left-to-right undulating “slinky-like” motion of the hits histogram for generations 0, 10, 20, 30, 40, and 46 is shown below.

4.8.2. CASE II

The environment consists of 10 initial environmental starting condition cases. Position is chosen randomly between -0.5 and +0.5 meters. Velocity v is chosen randomly between -0.5 and +0.5 meters/second. Angle θ is chosen randomly between -0.5 radians (about 28.6 degrees) and +0.5 radians. Angular velocity ω is chosen randomly between -0.5 and +0.5 radians/second. The force F is 4.0 Newtons.

Note that in this version of this problem, the angle θ lies between -28.6 degrees and 28.6 degrees and that the angular velocity ω lies between -0.5 and +0.5 radians/second. When the angle θ is limited to a domain of about -11.5 degrees to +11.5 degrees, Sin θ approximately equals θ. In this limited domain, the solution to the problem becomes somewhat simplified, and, in some cases, a simple hyperplane can be a reasonably acceptable control surface. However, the domains of the state variables we use here put the problem clearly into the non-linear area.

Time was discretized into 400 time steps of .02 seconds. The total time available before the system "times out" for a given control strategy is thus 8 seconds. If the square root of the sum of the squares of the velocity v, angle θ , and angular velocity ω (i.e. the "norm") is less than 0.07 (the "criterion"), the system is considered to have arrived at its target state (i. e. the broom balanced and the cart at rest). If a particular control strategy brings the system to the target state for a particular environmental starting condition case, its fitness for that environmental case is the time required (in seconds). If a control strategy fails to bring the system to the target state before it "times out", its fitness for that environmental case is set to 8 seconds. The "fitness" of a control strategy is the average time for the strategy over all 10 environmental cases.

The initial population of random control strategies in generation 0 includes many highly unfit control strategies, including totally blind strategies that ignore all the state variables, partially blind strategies that ignore some of the state variables, strategies that repetitively apply the force from only one direction, strategies that are correct only for a particular few specific environmental cases, strategies that are totally counter-productive, and strategies that cause wild oscillations and meaningless gyrations.

In one run, the average time consumed by the initial random strategies in the initial random population averaged 7.79 seconds. In fact, many of these 300 random individuals "timed out" at 8 seconds for all 10 of the environmental cases (and very likely would have timed out even if more time had been available). However, even in this highly unfit initial random population, some control strategies are somewhat better than others. The best single control strategy for the initial random generation was the simple non-linear control strategy $v\,\theta$, which averaged 6.55 seconds. This control strategy correctly handled two (of the simpler) environmental cases and timed out for 8 of the environmental cases. Notice that this control strategy is partially blind in that it does not even consider the state variable ω in specifying how to apply the "bang bang" force.

The genetic crossover operation is then applied to parents from the current generation selected with probabilities proportionate to fitness to breed a new population of offspring control strategies. Although the vast majority of the new offspring control strategies are again highly unfit, some of them tend to be somewhat more fit than others. Moreover, some of them are slightly better than those that came before.

For example, the average population fitness improved from 7.79 to 7.78, 7.74, 7.73, 7.70, 7.69, 7.66, 7.63, 7.62 seconds per environmental case in generations 1 through 8, respectively. This generally improving (but not necessarily monotonic) trend in average fitness is typical of genetic algorithms.

In generation 6, the best single individual was the non-linear control strategy $\theta > v^3 \left| \square\theta\square + \square v \right|$ It required an average of 6.45 seconds. Notice that this particular control strategy explicitly incorporated the greater-than function > in the strategy, whereas the earlier control strategy was merely a polynomial that defined a control surface.

In generation 7, the best single individual in the population correctly handled (i.e. did not time out) 4 of the l0 environmental cases. It required an average of 5.72 seconds.

In generation 8, the best single individual was the non-linear control strategy

$$[\theta + \omega + 2.762v + \theta^3]^3$$

It required an average of only 2.56 seconds. It correctly handles 10 of the 10 environmental cases. Note that the number of correctly handled environmental cases ("hits") is a statistic that we find useful for monitoring runs, but this statistic is not used by the genetic algorithm.

In generation 14, the average time required by the best single individual in the population dropped below 2 seconds for the first time. In particular, the non-linear control strategy

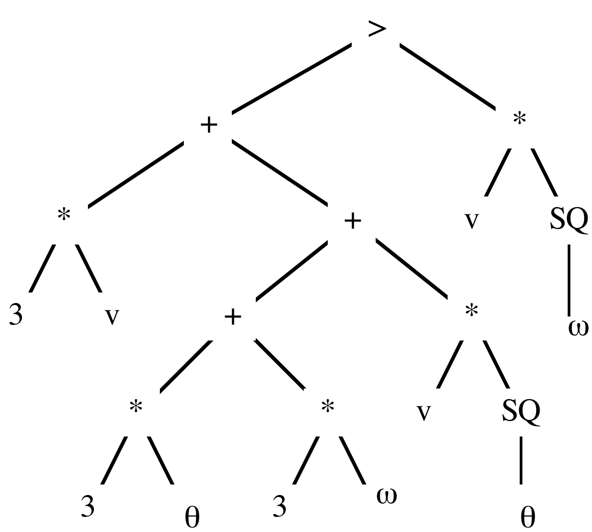
$$\theta + \omega + v + v^2 [\omega - 0.734]^2 > v \omega^2$$

required an average of only 1.74 seconds. It, too, correctly handled all l0 of the environmental cases.

Two histograms provide a graphical picture of the learning of the population from generation to generation. The "hits histogram" shows the number of individuals in the population that correctly handle a particular number of environmental cases. The "fitness histogram" shows the number of individuals in the population with whose fitness value lies in a particular decile of the fitness values. Both of these histograms have an undulating left-to-right "slinky-like" motion as the population progressively learns.

In generation 33, the best single individual in the population was the non-linear control strategy

$$3v + 3\theta + 3w + v\theta^2 > v\omega^2$$



After its discovery, this single best control strategy was retested on 1000 additional random environmental starting condition cases. It performed in an average of 1.76 seconds. In another test, it averaged 3.20 seconds on the 8 corners of the cube, but could not handle 2 of the 8 corners of the cube.

The benchmark pseudo optimum strategy averaged 1.85 seconds over the 1000 random environmental cases in the retest. It averaged 2.96 seconds for the 8 corners of the cube. It was unable to handle the 2 worst corners of the cube.

These results (in seconds) are summarized in the table below:

CASE II - PERFORMANCE FOR 3-DIMENSIONAL BROOM BALANCING PROBLEM

Control Strategy	1000 Points	8 Corners	Worst 2 Corners
Benchmark Pseudo Optimum	1.85	2.96	Infinite
$3v + 3\theta + 3w + v\theta^2 > v\omega^2$	1.76	3.20	Infinite

There is no known solution for this problem nor is there any specific test we can perform on an apparent solution that we obtain to verify that it is the optimum. We do know that the control strategy

$$3v + 3\theta + 3w + v\theta^2 > v\omega^2$$

had the best time of those we discovered; that we discovered a number of other control strategies that were slightly worse (and hence this control strategy seemed to be the result of some kind of convergent process); and that it is slightly better than the benchmark psuedo optimum strategy.

4.9. FINDING GAME-PLAYING STRATEGIES

In a game, there are two or more independently-acting players who make choices (moves) and receive a payoff based on the choices they make. A "strategy" for a given player in a game is a way of specifying what choice (move) the player is to make at a particular point in the game from all the allowable moves at that time and given all the information about the state of the game that is available to the player at that time. Strategies for games may be expressed in several different ways. One conceptually simple, but often tedious, way is to specify a given player's moves in terms of every possible sequence of previous moves made by that player and his opponents. Another way is to express the strategy in terms of the state of the game or in terms of various features abstracted from the state of the game.

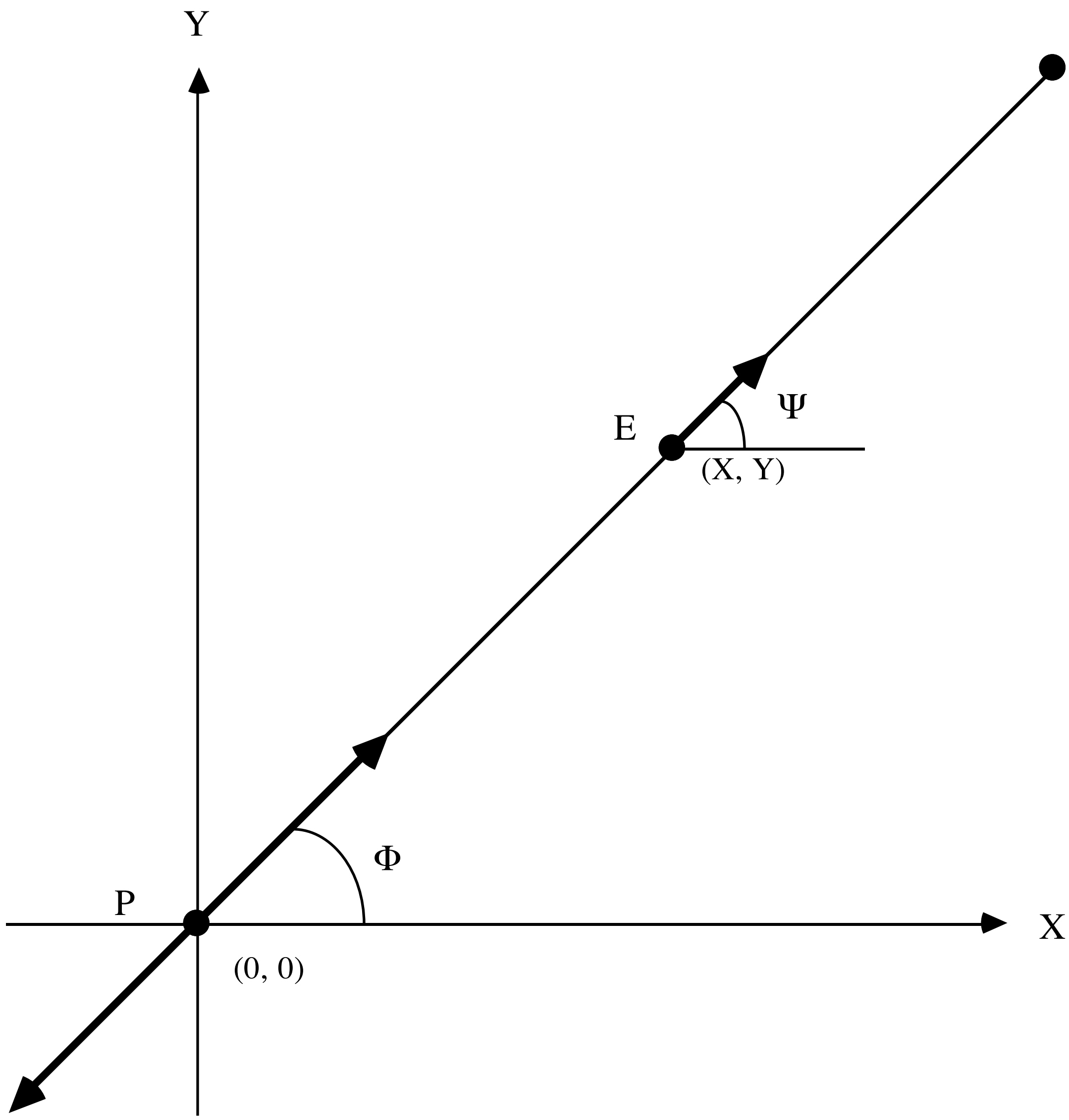
4.9.1. DIFFERENTIAL PURSUER-EVADER GAME

The "game of simple pursuit" described in Isaac's *Differential Games* (1965) is a 2-person, competitive, zero-sum, simultaneous-moving, complete-information game in which there is a pursuing player P which is trying to capture an evading player E. The “choices” available to a player at a given moment in time consist of choosing a direction (angle) in which to travel. In the simple game of pursuit, the players travel in a plane and both players may instantaneously change direction without restriction (i.e. either smoothly or abruptly). Each player travels at a constant speed, and the pursuing player’s speed w_p (1.0) is greater than the evading player’s speed w_e (0.67).

The state variables of the game are x_p, y_p, x_e, and y_e representing the coordinate positions (x_p, y_p) and (x_e, y_e) of the pursuer P and evader E in the plane.

At each time step, both players know the position (state variables) of both players. The choice for each player is to select a value of their control variable (i.e. the angular direction in which to travel). The pursuer’s control variable is the angle ϕ (from 0 to 2Π radians) and the evader’s control variable is the angle ψ . The players choose the value for their control variable simultaneously.

The analysis of this game can be simplified by reducing the number of state variables from four to two. This state reduction is accomplished by simply viewing the pursuer P as being at the origin point (0,0) of a new coordinate system at all times and then viewing the evader E as being at position (x, y) in this new coordinate system. The two numbers x and y representing the position (x, y) of the evader E thus become the two "reduced" state variables of the game. Whenever the pursuer P travels in a particular direction, the coordinate system is immediately adjusted so that the pursuer is repositioned back to the origin (0, 0) and then appropriately adjusting the position (x, y) of the evader to reflect the travel of the pursuer.



The state transition equations for the evader E are as follows:

$$x(t+1) = x(t) + \; w_e \mathrm{Cos}\, \psi \; - \; w_p \mathrm{Cos}\, \phi$$

$$y(t+1) = y(t) + \; w_e \mathrm{Sin}\, \psi \; - \; w_p \mathrm{Cos}\, \phi \, .$$

In order to develop optimal playing strategies, we use a set of random environmental starting condition cases consisting of N_e (= 10) starting positions (x_i, y_i) for the evader E. Each starting value of x_i and y_i is between -5.0 and +5.0. The two players may travel anywhere in the plane. We regard the pursuer P as having captured the evader E when the pursuer gets to within some specified small distance $\varepsilon = 0.5$ of the evader E.

The payoff for a given player is measured by time. The payoff for the pursuer P is the total time it takes to capture the evader E over all of the environmental cases. The pursuer tries to minimize this time to capture. The payoff for the evader E is the total time of survival for E. The evader tries to maximize this time of survival. In this particular simple game of pursuit, the pursuer's superior speed makes it possible for the pursuer to always capture the evader for any set of environmental cases.

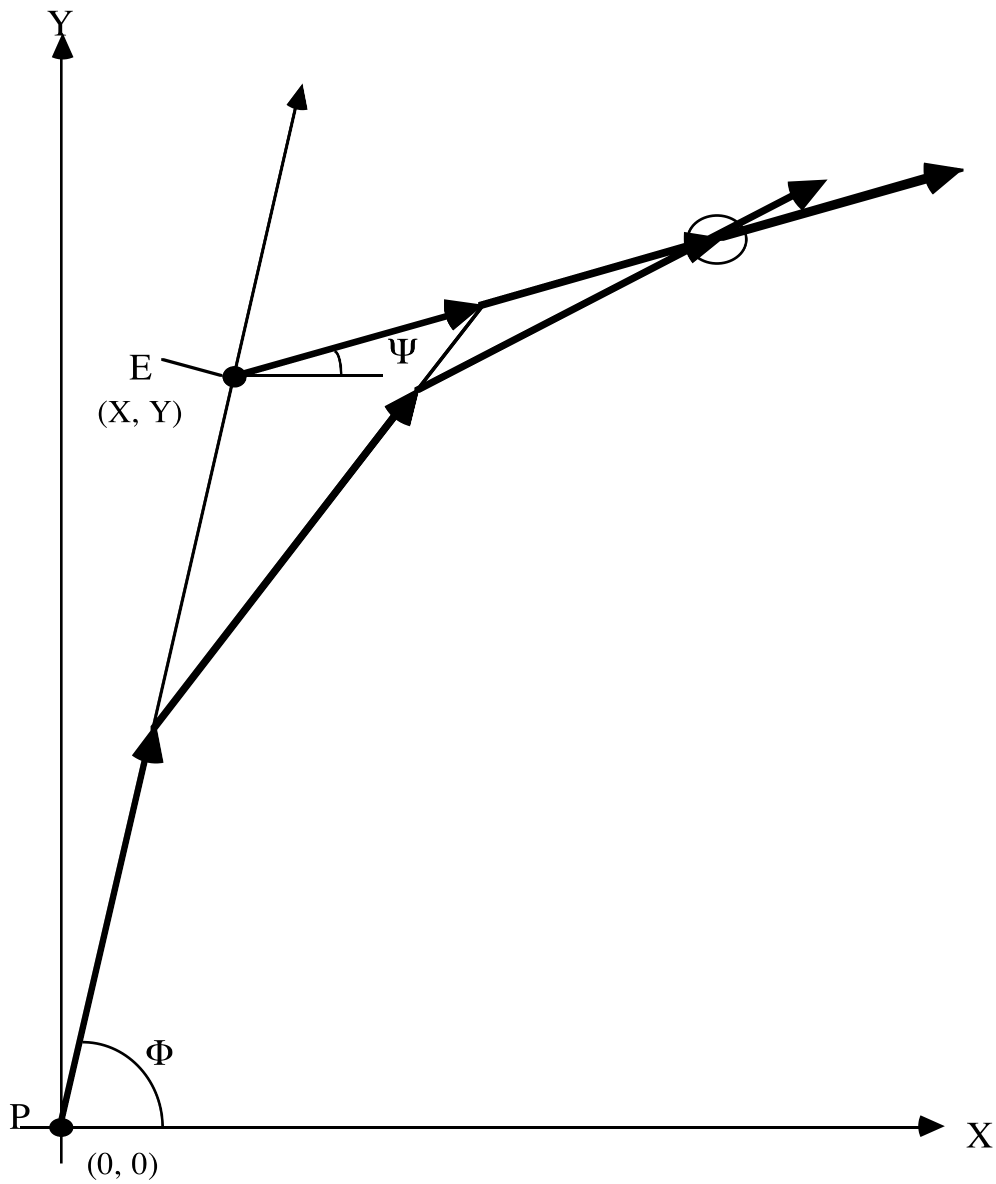
A maximum "time out" time (100 time steps) is established so that if the pursuer has not made the capture within that amount of time, that maximum time becomes the payoff for that particular environmental case and that particular strategy.

The problem is to find the strategy for choosing the control variable of the pursuer so as to minimize the total time to capture for any set of environmental cases when playing against an optimal evader which is trying to maximize his total time of survival for any set of environmental cases.

For this particular simple game, the best strategy for the pursuer P at any given time step is to chase the evader E in the direction of the straight line currently connecting the pursuer to the evader. And, for this particular simple game, the best strategy for the evader E is to race away from the pursuer in the direction of the straight line connecting the pursuer to the evader.

In comparison, the worst strategy for the pursuer P is to avoid the evader E by racing away from the evader in the direction precisely opposite to the straight line currently connecting the pursuer to the evader. The worst strategy for the evader E is to race towards the pursuer P along this same straight line.

If the evader chooses some action other than the strategy of racing away from the pursuer in the direction of the straight line connecting the pursuer to the evader, the evader will survive for less time than if he follows his best strategy. If the evader initially chooses a sub-optimal direction and then belatedly chooses the optimal direction, his time of survival is less than if he had chosen the optimal direction from the beginning.



The situation is symmetric in that if the pursuer does not chase after the evader E along the straight line, he fails to minimize the time to capture.

The “value of the game” is the payoff (time) such that, no matter what the evader does, the evader cannot hold out for longer than this amount of time, and, if the evader does anything other than direct fleeing, his survival time is a shorter amount of time. Conversely, no matter what the pursuer does, the pursuer P cannot capture an optimal evader E in less than the amount of time equal to the “value of the game”, and, if the pursuer does anything other than direct pursuit, the evader can remain at large for a longer amount of time.

The ”genetic programming paradigm paradigm” can be used to solve the differential game of simple pursuit by genetically evolving a population of pursuing individuals over a number of generations. The environment that consists of an optimal evader. In other words, each individual in the population of pursuing individuals is tested against the optimal evader so that the population of pursuing individuals could improve based on this testing (i.e. absolute fitness). Similarly, an optimal evader can be genetically evolved out of a population of evading individuals if the environment consists of an optimal pursuer.

The “genetic programming paradigm” is especially well suited to solving this kind of problem because the solution takes the form of a mathematical expression whose size and shape may not be known in advance and which may dynamically vary during the process.

Identification of the terminals for solving this problem using the “genetic programming paradigm” is straight-forward for this problem. The terminal set T contains the two reduced state variables X and Y representing the position of the evader E in the plane. That is, the terminal set is T = {X, Y, R}, where R is the emphemeral random constant.

The function set for this problem can be a set of arithmetic and mathematical operations such as addition, subtraction, multiplication, division (using the operation % which returns a zero when division by zero is attempted), and the exponential function EXP. Thus, the function set is {+, -, *, %, EXP}.

If the population of individuals represents pursuers and we are attempting to genetically breed an optimal pursuing individual, the environment for this “genetic algorithm” consists of an optimal evading individual. The optimal evader travels with the established constant evader speed w_e in the angular direction specified by the Arctangent function. The Arctangent function has two arguments (the X value and the Y value). The Arctangent function returns a number between 0 and 2 Π radians (360 degrees) representing the angle whose tangent is Y/X. If X happens to be zero, the sign of Y determines the value of the Arctangent function. In particular, the Arctangent function return Π /2 radians (90 degrees) if the sign of Y is positive and it returns

3 π /2 radians (270 degrees) if the sign of Y is negative. The Arctangent function returns 0 in the event that both X and Y are zero.

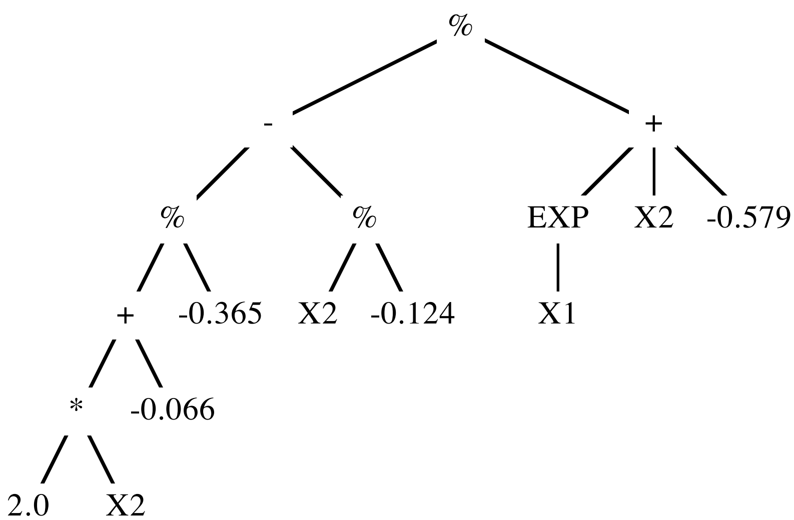
We use a set of random environmental starting condition cases consisting of N_e (= 10) starting positions (x_i, y_i) for the evader E. Each starting value of x_i, y_i is a random number between -5.0 and +5.0.

As one progresses from generation to generation, the population of pursuing individuals typically improves. After several generations, the best pursuing individuals in the population can capture the evader in a small fraction (perhaps 2, 3, or 4) of the 10 environmental cases within a certain amount of time. Then, after additional generations, the population improves and the best pursuing individuals in the population can capture the evader in a larger fraction (perhaps 4, 5, or 6) of the 10 environmental cases within a shorter amount of time. Often, these partially effective pursuers are effective in some identifiable fraction of the plane or at some identifiable range of distances, but ineffective in other parts of the plane or at other distances. However, as more and more generations are run, the population of pursuing individuals typically continues to improve.

In one run, a pursuer strategy emerged in the 17th generation which correctly handled all 10 of the environmental cases. This S-expression is shown below:

```
(% (- (% (+ (* 2.0 X2) -0.066) -0.365) (% X2 -0.124))
   (+ (EXP X1) X2 -0.579)).
```

This S-expression is depicted graphically below:



This S-expression is equivalent to

$$\frac{\frac{2x_2 - 0.066}{-0.365} + \frac{x_2}{0.124}}{e^{x_1} + x_2 - 0.579}$$

which in turn is equivalent to

$$\frac{2.58x_2 + 1.81}{e^{x_1} + x_2 - 0.579}$$

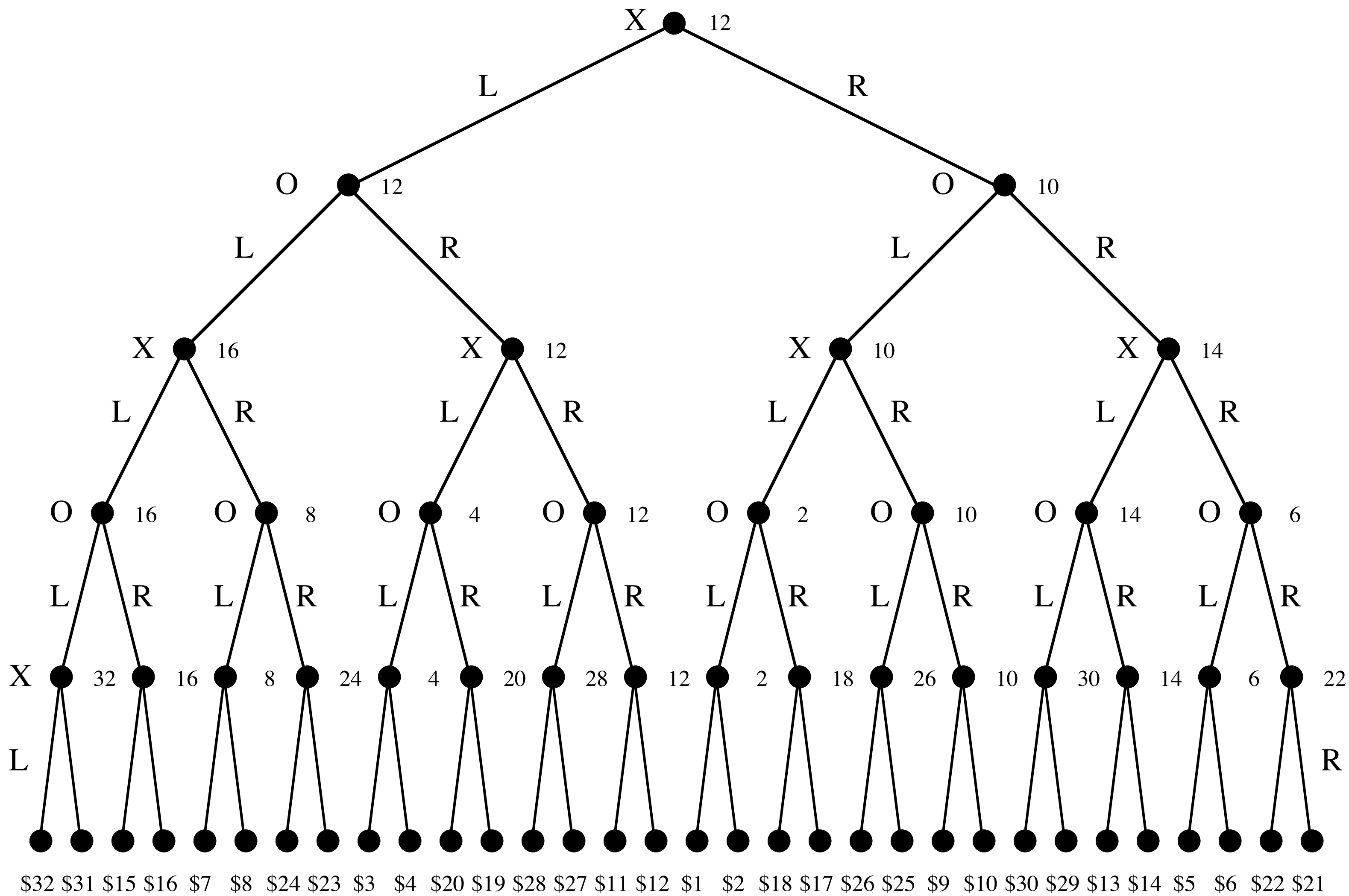
When this apparent optimal pursuing individual is re-tested against a much larger set of environmental cases (i.e. 100 or 1000), we then find that it also successfully handles 100% of the environmental cases. Thus, this S-expression is an optimal solution to the problem here.

Note also that the above apparent optimal solution is dependent on the particular values of the parameters of this problem and that slightly different pursuing individuals might emerge if other possible values of these parameters had been used. Because we intentionally did not include the Arctangent function in the function set, and because the MacLaurin infinite series for the Arctangent function (which uses multiplication, addition, subtraction, and division) converges only for a limited domain of argument values, there may be combinations of parameters for this problem for which these above apparent optimal solution may no longer be a solution to the problem. In particular, if the zone of capture parameter ε (currently 0.5) were substantially reduced (so as to require greater accuracy at the moment of capture), the range of environmental cases were substantially expanded (beyond the -5 to +5 range used here in each direction), or the “time out” time were substantially reduced, we might find a small fraction of environmental cases (out of a large number of environmental cases) for which the above apparent optimal pursuing individuals would not be successful.

A minimax strategy for the evader was discovered in a similar way.

4.9.2. DISCRETE GAME IN EXTENSIVE FORM

Consider the simple discrete game whose game tree in extensive form is shown below:


X 12
L R
O 12
O 10
L R L R
X 16
X 12
X 10
X 14
O 16
O 8
O 4
O 12
O 2
O 10
O 14
O 6
X 32 16 8 24 4 20 28 12 2 18 26 10 30 14 6 22
L R
$32 $31 $15 $16 $7 $8 $24 $23 $3 $4 $20 $19 $28 $27 $11 $12 $1 $2 $18 $17 $26 $25 $9 $10 $30 $29 $13 $14 $5 $6 $22 $21

This game is a "2-person" “competitive” “zero-sum” game in which the players make alternating moves. Player X starts and can choose to go L (left) or R (right). Then, player O can choose to go L (left) or R (right). After player X has made three moves and player O has made two moves, player X receives (and player O pays out) the payoff shown at the particular endpoint (of 32) of the game tree that was reached.

Since this game is a game of complete information, each player has access to information about his opponent's previous moves (and his own previous moves). This historical information is contained in five variables XM1 ("X's move 1"), OM1 (O's move 1"), XM2 (X's move 2), OM2 (O's move 2), and XM3 (X's move 3). These five variables each assume one of three possible values: L (left), R (right), or U (undefined). A variable is undefined prior to the time when the move to which is refers has been made. For example, at the beginning of the game, all five variables are undefined.The particular variables that are defined or undefined indicate the point to which play has progressed during the play of the game. For example, if both players have moved once, XM1 and OM1 are defined (as either L or R) but the other three variables (XM2, OM2, and XM3) are undefined (i. e. have the value U).

A strategy for a particular player in a game specifies what move that player is to make for every possible situation that may arise for that player. In particular, in this game, a strategy for player X must specify his first move if he happens to be at the beginning of the game. A strategy for player X must also specify his second move if player O has already made one move and it must specify his third move if player O has made two moves. Since Player X moves first, player X's first move is not conditioned on any previous move. But, player X's second move will depend on Player O's first move (i.e. OM1) and, in general, his own first move (XM1). Similarly, player X's third move will depend on player O's first two moves and, in general, his own first two moves. Similarly, a strategy for player O must specify what choice player O is to make for every possible situation that may arise for player O. A strategy here is a computer program (i.e. LISP S-expression) whose inputs are the relevant historical variables and whose output is a move L (left) or R (right) for the player involved. Thus, the terminal set for this problem is T = {L, R}.

Four testing functions CXM1, COM1, CXM2, and COM2 provide the facility to test each of the historical variables that are relevant to deciding upon a player's move. Each of these functions is a specialized form of the CASE function in Common LISP. For example, function CXM1 has three arguments and evaluates it first argument if XM1 (X's move 1) is undefined, evaluates it second argument if XM1 is L (Left), and evaluates its third argument if XM1 is R (Right). Functions CXM2, CXM3, COM1, and COM2 are similarly defined. Thus, the function set for this problem is F = {CXM1, COM1, CXM2, COM2} with three arguments each.

Our first goal is to evolve a game-playing strategy for player X for this game. The minimax strategy for player O serves as the environment for evolving game-playing strategies for player X.

In one run, the best single individual game-playing strategy for player X in generation 6 had a minimax value of 88 and scored 4 minimax hits. This strategy was the S-expression

```
(COM2 (COM1 (COM1 L (CXM2 R (COM2 L L L) (CXM1 L R L))
                      (CXM1 L L R)) L R) L (COM1 L R R).
```

This S-expression simplifies to

```
(COM2 (COM1 L L R) L R).
```

If both OM2 (O's move 2) and OM1 (O's move 1) are undefined (U), it must be player X's first move. That is, we are at the beginning of the game (i.e. the root of the game tree). In this situation, the first argument of the COM1 function embedded inside the COM2 function of this strategy specifies that player X is to move L. The left move by player X at the beginning of the game is player X's minimax move because it takes the game to a point with minimax value 12 (to player X) as opposed to a point with only minimax value 10.

If OM2 (O's move 2) is undefined but OM1 is defined, it must be player X's second move. In this situation, this strategy specifies that player X moves L if OM1 (O's move 1) was L and player X moves R if OM1 was R. If OM1 (O's move 1) was L, player O has moved to a point with minimax value 16. Player X should then move L (rather than R) because that move will take the game to a point with minimax value 16 (rather than 8). If OM1 was R, player O has moved to a point with minimax value 12. This move is better for O than moving L. Player X should then move R (rather than L) because that move will take the game to a point with minimax value 12 (rather than 4).

If both OM1 and OM2 are undefined, it must be player X's third move. If OM2 was L, player X can either choose between a payoff of 32 or 31 or between a payoff of 28 or 27. In either case, player X moves L. If OM2 was R, player X can choose between a payoff of 15 or 16 or between a payoff of 11 or 12. In either case, player X moves R. In this situation, this S-expression specifies that player X moves L if OM2 (O's move 2) was L and player X moves R if OM2 was R.

If player O has been playing its minimax strategy, this S-expression will cause the game to finish at the endpoint with the payoff of 12 to player X. However, if player O was not playing his minimax strategy, this S-expression will cause the game to finish with a payoff of 32, 16, or 28 for player X. The total of the 12, 32, 16, and 28 is 88 and the attainment of these four values constitutes 4 minimax hits.

We then proceeded to to evolve a game-playing strategy for player O for this game. The minimax strategy for player X serves as the environment for evolving game-playing strategies for player O.

In one run, the best single individual game-playing strategy for player O in generation 9 had a minimax value of 52 and scored 8 minimax hits. This strategy was the S-expression

```
(CXM2 (CXM1 L (COM1 R L L) L) (COM1 R L (CXM2 L L R))
      (COM1 L R (CXM2 R (COM1 L L R) (COM1 R L R)))).
```

This S-expression simplifies to

```
(CXM2 (CXM1 # R L) L R),
```

where the symbol # indicates the response to a situation that cannot ever arise.

4.10. PROBLEMS USING STRUCTURE-PRESERVING CROSSOVER

In this section, we present two problems that can be handled more naturally and efficiently if it is recognized that the individuals in the population inherently have a more complex structure than an unrestricted S-expression and that this structure should be preserved as crossover occurs.

4.10.1. MULTIPLE REGRESSION

In the earlier examples of symbolic regression and empirical discovery (e.g. econometric time series, Kepler's Law), there was always one dependent variable (with perhaps several independent variables). Problems with more than one dependent variable can also be genetically solved. This problem illustrates the need for creating random random individuals that comply with a particular set of syntactic restrictions and then performing structure-preserving crossover on those individuals.

Consider a problem with four independent variables x_1, x_2, x_3, and x_4 and two dependent variables y_1 and y_2. Suppose we are given a set of 50 data points in the form of 50 6-tuples of the form $(x_{1i}, x_{2i}, x_{3i}, x_{4i}, y_{1i}, y_{2i})$). The unknown function relationships are

$$y_{1i} = x_{1i}x_{3i} - x_{2i}x_{4i}$$

$$y_{2i} = x_{2i}x_{3i} + x_{1i}x_{4i}$$

for i between l and 50.

Two changes are required from the previously described process in order to most naturally and efficiently solve this problem.

First, the root of tree (i.e. the function just inside the left-most parenthesis of the LISP S-expression) should be the function LIST with two arguments. That is, the value returned by the S-expression is the list (vector) of two numbers, rather than a single number.

The LISP S-expression created in the initial generation can have this desired structurally merely by restraining the choice for the function for the root of the tree (i.e. the function occurring just inside the left-most parenthesis of the S-expression) to the function LIST with two arguments. Thereafter, the choice of function for any other internal point of the tree is an unrestricted choice from amongst the other functions of the problem (e.g. addition, subtraction, multiplication, etc. here).

Second, the choice of points in the crossover operation should be restrained so as to preserve the structure just created above for the initial generation. In particular, the structure for this problem is that each individual in the population must have the function LIST as its root and must not have the function LIST elsewhere in the tree.

This restraining process can be viewed in two ways. The simplest (and least general way) to view this restraining process is to exclude the root of the tree from being selected as the crossover point of either parent in the crossover operation. With this exception, the crossover operation then proceeds as usual. This restraint guarantees that the crossover operation preserves the structure required by this particular problem.

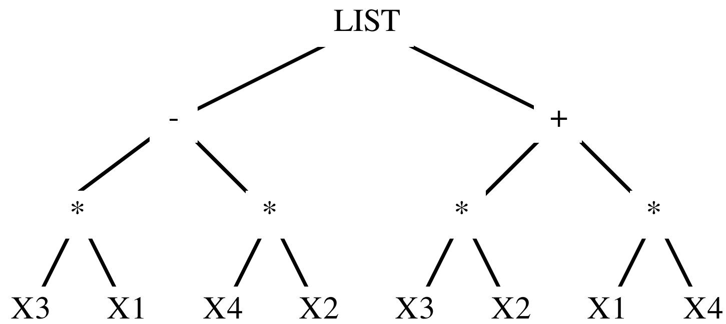
A more general way of viewing this restraining process for the crossover operation illustrates the general principle involved in structure-preserving crossover in a more generally applicable way. Any point may be selected as the crossover point for the first parent. There is no restriction. However, the selection of the crossover point in the second parent is then restricted to a point of the same "type" as the point just chosen from the first parent. For the multiple regression problem, there are only two "types" of points involved. The root of the tree is one "type" and all other points of the tree are the second "type." Thus, if a non-root points is chosen as the crossover point for the first parent, then a non-root point must be chosen as the crossover point of the second parent. If the root happens to be chosen as the crossover point for the first parent, then the selection of the crossover point of the second parent is restrained to points of this same "type." For this particular problem, there is only one point in the second parent that is of this same type, namely, the root point of the second parent. As it happens, when the crossover operation selects the roots both parents as crossover points, the operation merely swaps the entire parental trees. That is, the crossover operation degenerates into an instance of fitness proportionate reproduction in this special case. The fact that this approach is the more generally applicable way of viewing the restraining process will become clearer below in connection with neural network design.

When multiple values are returned by an S-expression, the fitness function must be also be modified. One way to do this is to make the fitness equal to the absolute value of the difference between the value of the first dependent variable returned by the S-expression and the target value of the first dependent variable plus the absolute value of the difference between the value of the second dependent variable returned by the S-expression and the target value of the second dependent variable. Of course, other ways of measuring distances (such as the square root of the sum of the squares of the differences) can also be used.

The set of terminals for this problem is T = {X1, X2, X3, X4}.

In one run, the best LISP S-expression in the 31st generation was

```
(LIST (- (* X3 X1) (* X4 X2))  (+ (* X3 X2) (* X1 X4))).
```



The two arguments to the LIST function are the two desired functional relationships. Note that complex multiplication is implemented by these return values.

The technique of handling the return of more than one value from a program and the associated calculation of fitness in that situation can be applied to many other problems. For example, two values may be returned as the output of a neural network; or, multiple values returned from a program that performs a task.

4.10.2. NEURAL NETWORK DESIGN

In this experiment, we show how to simultaneously design a neural network for both its weights and its architecture (namely, the number of layers in the neural net, the number of processing elements in each layer, and the connectivity between processing elements).

Neural networks are networks of directed lines containing linear threshold processors at the internal points in the network and numerical weights on each of the non-output lines. In a neural network, the inputs are those external points that are connected to a directed line segment pointing towards an internal point of the network and the outputs are those external points that are connected to a directed line segment pointing out from an internal point of the network. The signal on each non-output line segment is multiplied by a numerical constant (called the weight) associated with that line as it passes along that line segment. The linear threshold processing elements at the internal points of the network emit a signal from a discrete set of possibilities (i.e. 0 or 1). If the sum of the weighted inputs to the processor exceeds a certain threshold value T, the processor emits a 1 signal. Otherwise, it emits a 0 signal. That is, the output Oj of the linear threshold processing element j in a neural network is

$$\begin{cases} 1 & \text{if } \sum_{i=0}^{N} w_{ij} S_{ij} + b_j > T_j \\ 0 & \text{otherwise} \end{cases}$$

where S_{ij} is the i-th input signal to processing element j, where w_{ij} is the i-th weight to processing element j, where T_j is the threshold for processing element j, and where b_j is the bias (if any) of processing element j. Without loss of generality, we assume that the thresholds of all the neural network processing elements are 1.0 and that there are no biases in the network (although it is clear that the thresholds and biases could also be genetically discovered in the same general way as described below).

A wide variety of different neural network architectures are described in the literature (Hinton 1989). Most currently studied neural networks have at least one intermediate, hidden layer of processing elements lying between the inputs and the processing elements that are directly connected outputs (Rumelhart, Hinton, and Williams 1986). This contrasts with early single layer "perceptrons" where there was no hidden layer of processing element (Minsky and Papert 1969). Most neutral networks discussed in the current literature are feed-forward only, although the so-called recurrent neural networks allow feedback.

There are a wide variety of different neural network training paradigms described in the literature. Neural network training paradigms almost always presuppose that the architecture of the neural network has already been determined. That is, they presuppose that a selection has been made for the number of layers of linear threshold processing elements, the number of processing elements in each layer, and the nature of the allowable connectivity between the processing elements. In most neutral network training paradigms discussed in the current literature, the training paradigm involves exposing the neural network to "training cases" during a "training phase" and progressively modifying the weights (and sometimes the thresholds or biases) in some way that depends on how well the network's output matches the desired correct output associated with each training case. The goal is typically to have the neural network learn to perform some task so that it can correctly perform the task when it encounters repetitions, and, more importantly, previously unseen inputs cases of the task.

The function set for the neural network design problem contains at least six functions. In particular, the function set is F = {P, W, +, -, *, %}. The function P is the linear threshold processing function. The function W is the weighting function used to give a weight to a single going in to a linear threshold processing function. This weighting function W is, in fact, merely multiplication ; however, we give it this special name to distinguish it from the ordinary arithmetic operation of multiplication (which is used for the entirely different purpose of creating and modifying the numerical constants in the neural network in this problem). Both he linear threshold processing function P and the weighting function W typically appear in the function set with a varying number of arguments (i.e. perhaps 2, 3, 4, etc.). The function set also contains the four arithmetic operations addition (+), subtraction (-), multiplication (*), and restricted division function (%). The four arithmetic operations are used to create and modify the numerical constants (weights) of the neural network.

The set of terminals (arguments) contains the input signals of the problem and the ephemeral random constant atom ("R"). If there were two input signals (D0 and D1) for a particular problem, then the terminal set would be T = {D0, D1, R}.

Not all possible compositions of the functions from the function set above and terminals from the terminal set above correspond to what we would choose to call a "neural network." Thus, the problem of designing neural networks requires rules of construction that specify what structures are allowable for this particular problem. S-expressions must be initially created in conformity to these rules of construction and crossover must thereafter be performed in a structure-preserving way.

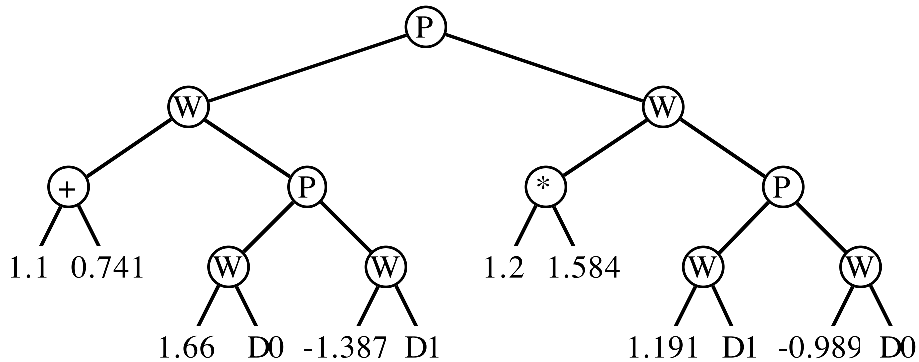
In particular, the rules of construction for neural networks with one output signal require that the root of the tree must be a processing element function P. Then, the functions at the level immediately below any processing element function P must be weighting functions (W). At the level below a weighting function, there is greater flexibility. In particular, there can be an input data signal (such as D0 and D1), any arithmetic function (addition, subtraction, division, or multiplication), a random constant, or another P function. At the level immediately below an arithmetic operation, there can only be an arithmetic operation or a random constant. Once a P function again appears in the tree, the rules of construction again require that the functions immediately below the P function again be weighting functions W. These rules are applied recursively until a full tree is constructed. Note that the external points of the tree are either input signals (i.e. D0 or D1) or random constants.

These rules of construction produce a tree (S-expression) that we would call a "neural network." The structure consists of linear threshold processing elements (the "P" functions) that process weighted inputs to produce a discrete signal (i.e. 0 or 1) as its output. The number of inputs to a processing element (P function) can vary; however, the inputs are always weighted signal lines. The weights may be a single numeric constant or can be the result of a composition of arithmetic operations and random constants. The signal lines can be the input to the network from the outside, the output of other processing elements, or compositions of arithmetic operations and random constants. If a signal line consists only of a composition of arithmetic operations and random constants, it is called a bias.

Consider the following LISP S-expression representing a neural network which l00% correctly performs the exclusive-or task on the inputs D0 and D1:

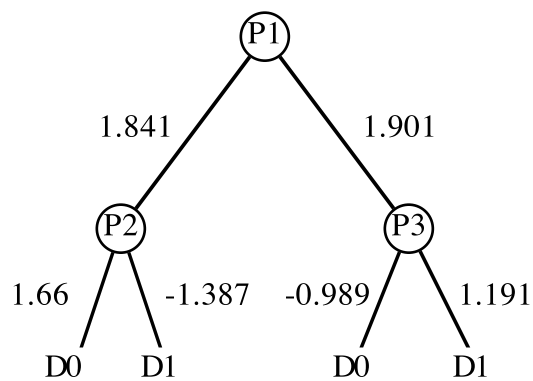
```
(P (W (+ 1.1 0.741) (P (W 1.66 D0) (W -1.387 D1)))
   (W (* 1.2 1.584) (P (W 1.191 D1) (W -0.989 D0)))).
```

This S-expression can be graphically depicted as the a rooted tree below:



The root of this tree contains the linear threshold processing function P with two arguments. The functions at the level immediately below the P function are both multiplication (*) functions. The left argument of the left-most multiplication function is a random constant 1.841. The right argument of the left-most multiplication function is the output of another linear threshold processing function P with two arguments. The functions at the level immediately below this second P function are both multiplication (*) functions. The arguments to the left multiplication function of this second P function are the random constant 1.66 and the input data signal D0. The arguments to the right multiplication function of this second P function are the random constant -1.387 and the input data signal D1.

This S-expression can be converted to the form that one would typically see in the neural network literature to the following:



Here the input signal D0 is weighted by 1.66 and the input signal D1 is weighted by -1.387 so that these two weighted input signals become the input to the linear threshold processing function P2 with two input lines. Since the input D0 is either 0 or 1, the first input line to P2 is either 0 or 1.66. Similarly, the second input line to P2 is either 0 or -1.387. The linear threshold processing function P2 adds up its two weighted input lines and emits a 1 if the sum exceeds the threshold of 1.0 and emits a 0 otherwise. If D0 and D1 are both 0, the sum of the inputs will be 0 (which is less than the threshold of 1) and therefore, P2 will emit 0. If D0 is 1 and D1 is 0, the sum will be 1.66 and P 2 will emit a 1. If D0 is 0 and D1 is 1, the sum will be -1.387 and P2 will emit a 0. If both

D0 and D1 are 1, the sum will be 0.273 . This is less than the threshold of 1.0. Thus, P2 will emit a 0. In other words, P2 emits a 1 if and only if the input lines are 10.

Similarly, the input signal D0 is weighted by -0.989 and the input signal D1 is weighted by 1.191 so that these two weighted input signals become the input to the linear threshold processing function P3 with two input lines. As will be seen, P3 will emit a 1 if and only if the input signals are 01.

Then, the output of P2 is weighted by 1.841 and the output of P3 is weighted by 1.901 so that these two weighted input signals become the input to the linear threshold processing function P1 with two input lines. As will be seen, the effect of these weights is that the weighted sum of the inputs to P1 exceeds the threshold of 1.0 if and only if either signals coming out of P2 and P3, or both, are non-zero.In other words, the output of P1 is 1 if either (but not both) D0 or D1 are 1, and the output of P is 0 otherwise.

That is, the output P1 is the exclusive-or function on the inputs D0 and D1.

If there is more than one output signal from the neural network, the output signals from the neural network are returned as a LIST in the manner described above for multiple regression. That is, the function set is enlarged to F = {LIST, P, W, +, -, *, %}. The rules of construction for neural networks with more than one output signal require that the root of the tree must be a LIST function. The number of arguments to the function LIST equals the number of output signals. This is the only time the LIST function is used in the S-expression. Then, the rules of construction require that the function at the level of the tree immediately below the LIST function must be linear threshold processing element functions P. Thereafter, the previously described rules of construction apply.

For example, the following LISP S-expression represents a neural network with two output signals:

```
(LIST (P (W D1 -1.423) (W D0 (+ 1.2 0.4))
     (P (W D0 (* -1.7 -0.9)) (W D0 (- 1.1 0.5)))).
```

The first argument to the LIST function is shown in underlined bold type. This first argument represents the first of the two output signals. This first output signal is the output from one linear threshold processing element. This linear processing element has two input lines. The first input line is connected to data signal line D1 and is weighted (multiplied) by -1.423 as it comes into the linear threshold processing element. The second input line is connected to input data signal line D0 and is weighted by the sum (+ 1.2 0.4) as it comes into the linear threshold processing element. The second argument to the LIST function is the second output signal and has a similar interpretation.

Note that a given input data signal (such as D0 or D1) can appear in more than one place in the tree. Thus, it is possible to create connectivity between any input data signal and any number of processing elements. Similarly, the "define building block" operation (section 7.4 below) makes it possible to have connectivity between the output from any linear processing element function P and the inputs of other linear processing element functions. The connectivity created using the "define building block" operation can even create feedback (i.e. a recurrent neural network); however, if only a feed-forward neural networks is desired, an additional rule of construction could implement that restriction.

In what follows, we show how to simultaneously find both the "architectural design" and the "training" of the neural network.

As an example, we seek to design a neural network to perform the task of adding two one-bit inputs to produce two output bits.

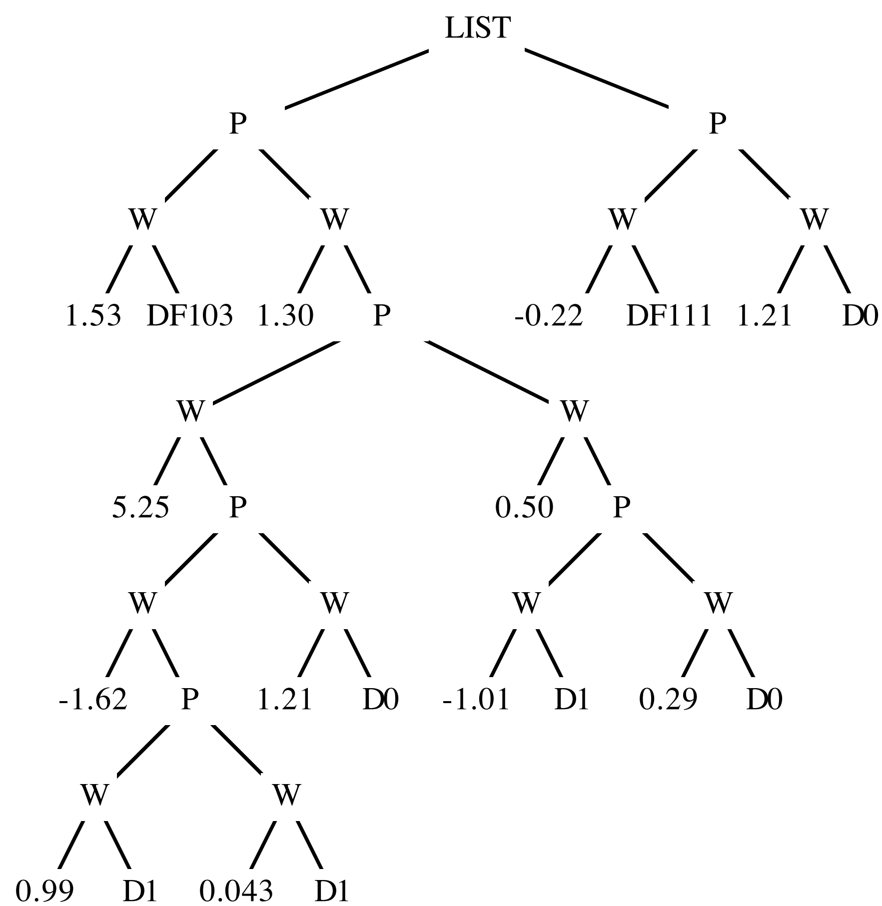
The environment for the task of adding two one-bit inputs to produce two output bits consists of the four cases representing the four combinations of binary input signals that could appear on D0 and D1 (i.e. 00, 01, 10, and 11). The correct outputs (00, 01, 01, and 10, respectively) are then associated with each of these four environmental cases (i.e. 00, 01, 10, and 11). The raw fitness function would add up the differences between the output signals from the neural network and the correct value of the one-bit adder function.

The crossover operation selects the crossover point in the first parent without restriction. However, once the crossover point is selected for the first parent, the crossover point for the second point must be of the same "type." For this problem, there are four "types" of points, namely (1) a point containing a processing element function P, (2) a point containing a weighting function W, (3) a point containing an input data signal (such as D0 or D1), and (4) a point containing an arithmetic operation or a random constant.

In one run, an individual emerged on generation 31 which l00% correctly performs the one-bit adder task. This individual was simplified by consolidating each sub-expression consisting of only numerical constants and arithmetic operations into a single numerical constant (weight).

```
(LIST
 (P (W 1.53 (DF103))
      (W 1.30
         (P (W 5.25 (P (W -1.62 (P (W 0.99 D1) (W 0.043 D1)))
                          (W 1.21 D0)))
              (W 0.50 (P (W -1.01 D1) (W 0.29 D0))))))
 (P (W -0.22 (DF111)) (W 1.21 D0))).
```

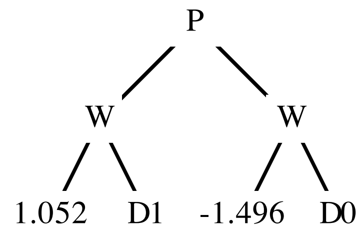
This individual is graphically depicted below.



Note that this S-expression contains two defined functions (DF103 and DF111). Defined function 103 is defined so as to return the S-expression shown below:

```
DF103 = (P (W 1.052 D1) (W -1.496 D0))
```

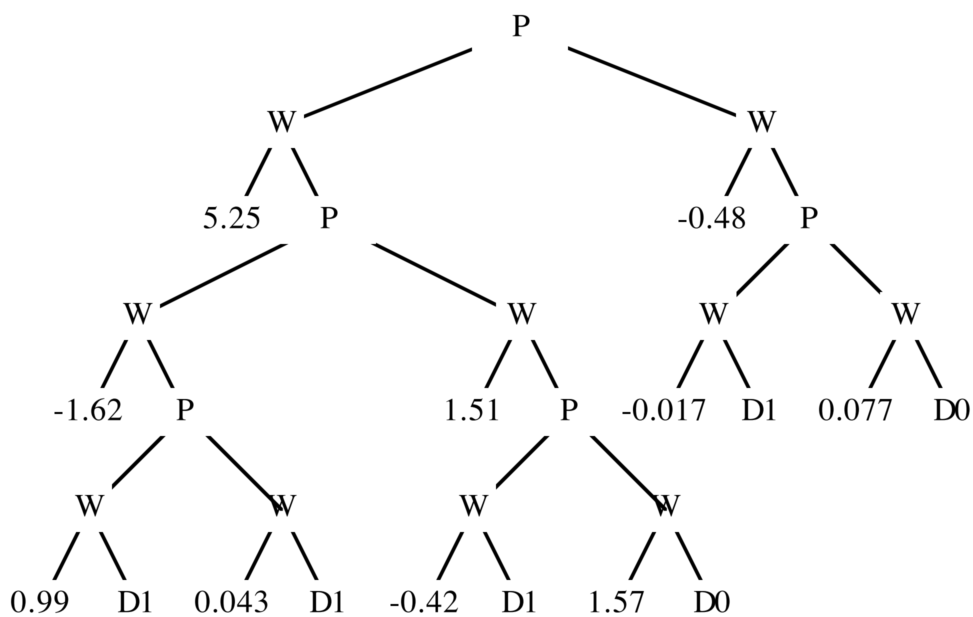
Defined function DF103 is graphically depicted below:



Similarly, defined function DF111 is defined so as to return the S-expression below:

```
DF111 =
(P
 (W 5.25
    (P (W -1.62 (P (W 0.99 D1) (W 0.043 D1)))
          (W 1.51 (P (W -0.42 D1) (W 1.57 D0)))))
 (W -0.48 (P (W -0.017 D1) (W 0.077 D0))))
```

Defined function 111 is graphically depicted below:



If the two defined functions (DF103 and DF111) are then inserted into the S-expression, we obtain the S-expression below. In this S-expression, the expanded version of the two defined functions are shown in bold type.

```
(LIST
  (P (W 1.53 (P (W 1.052 D1) (W -1.496 D0)))
      (W 1.30
          (P (W 5.25 (P (W -1.62 (P (W 0.99 D1) (W 0.043 D1)))
                          (W 1.21 D0)))
                (W 0.50 (P (W -1.01 D1) (W 0.29 D0))))))
  (P (W -0.22 (P
                  (W 5.25
                       (P (W -1.62 (P (W 0.99 D1)
                                            (W 0.043 D1)))
                            (W 1.51 (P (W -0.42 D1)
                                         (W 1.57 D0)))))
                  (W -0.48 (P (W -0.017 D1) (W 0.077 D0)))))
      (W 1.21 D0)))
```

In the above S-expression, the first element of the LIST is the low order bit of the result. Upon examination, this first element is equivalent to (OR (AND D1 (NOT D0)) (AND D0 (NOT D1))), which is the exclusive-or function of the two input bits D0 and D1. This is the correct expression for the low order bit of the result.

The second element of the LIST is the high order bit of the result. Upon examination, this second element is equivalent to (AND D0 (NOT (OR (AND D0 (NOT D1)) (NOT D1))), which is equivalent to (AND D0 D1). This is the correct expression for the high order bit of the result.

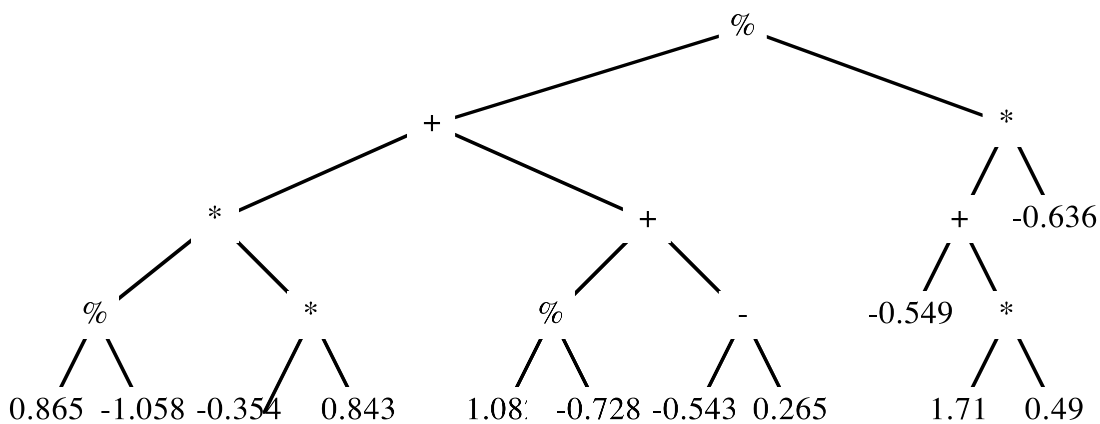
It is important to recall that each of the numeric constants in all of the S-expressions above is a consolidation of a sub-expression involving arithmetic operations and random constants. To illustrate this, we have highlighted one of the numeric constants (weights), namely the 5.25 occurring approximately in the middle of the S-expression above by underlining it and placing it in bold type. This highlighted S-expression is shown below:

```
(LIST
 (P (W 1.53 (P (W 1.052 D1) (W -1.496 D0)))
     (W 1.30
        (P (W 5.25
                (P (W -1.62 (P (W 0.99 D1)
                                    (W 0.043 D1)))
                      (W 1.21 D0)))
              (W 0.50 (P (W -1.01 D1) (W 0.29 D0))))))
 (P (W -0.22
       (P (W 5.25
               (P (W -1.62 (P (W 0.99 D1)
                                  (W 0.043 D1)))
                     (W 1.51 (P (W -0.43 D1)
                                   (W 1.57 D0)))))
     (W -0.48 (P (W -0.017 D1)
                     (W 0.076 D0)))))
  (W 1.21 D0))).
```

The process of consolidating sub-expressions consisting of only numerical constants and arithmetic operations can be illustrated by considering the sub-expression that gave rise to the weight 5.25 (underlined in bold type above). This weight arose from the following S-expression:

```
(% (+ (* (% 0.865 -1.058) (* -0.354 0.843))
      (+ (% 1.082 -0.728) (- 0.543 0.265)))
   (* (+ -0.549 (* 1.71 0.49)) -0.636)).
```

This sub-expression for the numerical constant (weight) 5.25 is depicted graphically below:



5. SUMMARY OF HOW TO USE THE ALGORITHM

In this section, we summarize the six major steps necessary for using the "genetic programming" paradigm. These majors steps involve determining (1) the set of terminals, (2) the set of function, (3) the environmental cases, (4) the fitness function, (5) the parameters for the run, and (6) the termination criterion and method for identifying the solution.

5.1. IDENTIFYING THE SET OF TERMINALS

The first major step is to identify the set of terminals for the problem. The set of terminals must, of course, be sufficient to solve the problem. The step of of correctly identifying the variables which have explanatory power for the problem at hand is common to all science. For some problems, this identification may be simple and straightforward. For example, in the broom-balancing problem, the physics of the problem dictate that the velocity of the cart, the angle of the broom, and the angular velocity of the broom are the state variables having explanatory power for the problem. In the sequence induction problem, the sequence index II is the single necessary variable atom (terminal). Needless to say, the set of variables must be sufficient to express the solution to the problem. For example, if one were given only the diameter of each planet and the color of its surface, one would not be able to discover Kepler's Third Law for the period of the planet.

Constant atoms, if required at all, can enter a problem in two ways. One way is to use the "constant creation" procedure involving the ephemeral random constant atom "R" described earlier. In this event, the type of such random initial constants is chosen to match the problem. For example, in a Boolean domain, the constants are T and NIL; in an integral domain, the constants are integers in a certain range; and in a real-valued problem domain, the constants might be floating point values in a certain range. The second way for constant atoms to enter a problem is by explicitly including them. For example, one might include π in a particular problem where there is a possibility that this particular constant would be useful. Of course, if one failed to include π in such a problem, the genetic programming paradigm paradigm would probably succeed in approximately creating it (albeit at a certain cost in computational resources) in the manner described above.

5.2. IDENTIFYING THE FUNCTION SET

The second major step is to identify a sufficient set of functions for the problem. For some problems, the identification of the function set may be simple and straightforward. For real-valued domains, the obvious function set might be the set of 4 arithmetic operations, namely, {+, -, *, %}. In a Boolean function learning domain, for example, the function set {AND, OR, NOT, IF} might be the choice since it is computationally complete and convenient (in that the IF function often produces easily understood logical expressions). If ones interests lie in the domain of design of semiconductor logic layout, a function set consisting only of the NAND function might be most convenient.

If the problem involves economics (where growth rates and averages often play a role), the function set might also include an exponential, logarithmic, and moving average function in addition to the four basic arithmetic operations. Similarly, the SIN and COS functions might be useful additions to the function set for some problems.

Some functions may be added to the function set merely because they might possibly facilitate a solution (even though the same result could be obtained without them). For example, one might include a squaring function in certain problems (e.g. broom balancing) even though the same result could be attained from the simple multiplication function (albeit at a cost in computational resources).

In any case, the set of functions must be chosen so that any composition of the available functions is valid for any value that any available variable atom might assume. Thus, if division is to be used, the division function must be modified so that division by zero is well-defined. The result of a division by zero could be defined to be zero, a very large constant, or a new value such as the Common LISP keyword ":infinity". If one defined the result of a division by zero as the keyword ":infinity," then, each of the other functions in the function set must be written so that it is well-defined if this ":infinity" value happens to be one of its arguments. Similarly, if square root is one of the available functions, it could either be a specially defined real-valued version that takes the square root of the absolute value of the argument (as was used in the broom balancing problem) or it could be the Common LISP complex-valued square root function SQRT (as was used in the quadratic equation problem).

Common LISP is quite lenient as to the typing of variables; however, it does not accommodate all of the combinations of types that can arise when computer programs are randomly generated and recombined via crossover. For example, if logical functions are to be mixed with numerical functions, then some kind of a real-valued logic should be used in lieu of the normal logical functions. For example, the greater than function GT used in the broom balancing problem assumed the real value 1.0 if the comparison relation was satisfied and the real value 0.0 otherwise.

Note that the number of arguments must be specified for each function. In some cases, this specification is obvious or even mandatory (e.g. the Boolean NOT function, the square root function). However, in some cases (e.g. IF, multiplication), there is some latitude as to the number of arguments. One might, for example, include a particular function in the function set with differing numbers of arguments. The IF function with two arguments, for example is the IF-THEN function, whereas the IF function with three arguments is the IF-THEN-ELSE function. The multiplication function with three arguments might facilitate the emergence of certain cross product terms although the same result could be achieved with repeated multiplication function with two arguments. It is often useful to include the Common LISP PROGN ("program") form with varying number of arguments in a function set to act as a connective between the unknown number of steps that may be needed to solve the problem.

The choice of the set of available functions, of course, directly affects the character of the solutions that can be attained. The set of available function form a basis set for generating potential solutions. For example, if one does symbolic regression on the absolute value function on the interval [-1, +1] with a function set containing the If-Then-Else function and subtraction, one obtains a solution in the familiar form of a conditional test on x that returns either x or -x. On other hand, if the function set happens to contain COS, COS3 (i.e. cosine of 3 times the argument), COS5 (i.e. cosine of 5 times the argument) instead of the If-Then-Else function, one gets two or three terms of the familiar Fourier series approximation to the absolute value function. Similarly, we have seen cases where, when the exponential function (or the SIGMA summation operator) was not available in a problem for which the solution required an exponential, the first one or two polynomial terms of the Taylor series in the solution, in lieu of the missing e^x.

It should be noted that the necessary preliminary selection of appropriate functions and terminals is a common element of machine learning paradigms. For example, in using techniques in the ID3 family for inducing decision trees, the necessary preliminary selection of the set of available "attribute-testing" functions that appear at the nodes of the tree (and the exclusion of other possible functions) corresponds to the process of choosing of functions here. Similarly, if one were approaching the problem of the 16-puzzle using SOAR, the necessary preliminary selection of the set of 24 operators for moving tiles in the puzzle corresponds to the process of choosing of functions here. Similarly, if one were approaching the problem of designing a neural network to control an artificial ant, as Jefferson, Collins et. al. successfully did (1990), the necessary preliminary selection of the functions (turn-left, turn-right, sense, move) corresponds to the process of choosing of functions here.

Naturally, to the extent that the function set or terminal set contains irrelevant or extraneous elements, the efficiency of the discovery process will be reduced.

5.3. ESTABLISHING THE ENVIRONMENTAL CASES

The third major step is the construction of the environment for the problem. In some problems, the nature of the environment is obvious and straight-forward. For example, in sequence induction, symbolic function identification (symbolic regression), empirical discovery, and Boolean function learning problems, the environment is simply the value(s) of the independent variable(s) associated with a certain sampling (or, perhaps, the entire set) of possible values of the dependent variable(s). In some problems (e.g. block-stacking, broom-balancing), the environment is a set of "starting condition" cases. In some problems where the environment is large (e.g. block-stacking), a random sampling or a structured representative sampling can be used. For example, the environmental cases for the symbolic regression problem, equation involving problems, differential game problem, and broom balancing problem were randomly selected floating points numbers in a specified range.

5.4. IDENTIFYING THE FITNESS FUNCTION

The fourth major step is construction of the fitness function. For many problems, the fitness function is the sum of the distances (taken over all the environmental cases) between the point in the range space returned by the S-expression for a given set of arguments and the correct point in the range space. One can use the sum of the distances or the square root of the sum of the squares of the distances in this computation.

For some problems, the fitness function is not the value actually returned by the individual S-expression in the population, but some number (e.g. elapsed time, total score, cases handled, etc.) which is indirectly created by the evaluation of the S-expression. For example, in the broom balancing problem, raw fitness is the average time required by a given S-expression to balance the broom. The goal is to minimize the average time to balance the broom over the environmental cases. In the "artificial ant" problem, the score is the number of stones on the trail which the artificial ant successfully traverses in the allowed time. Since the goal is to maximize this score, the raw fitness is the maximum score minus the score attained by a particular S-expression. In the block stacking problem, the real functionality of the functions in an individual S-expression in the population is the side effect of the S-expression on the state of the system. Our interest focuses on the number of environmental starting condition cases which the S-expression correctly handles. That is, the goal is to maximize the number of correctly handled cases. Since raw fitness is to be defined so that the raw fitness is closer to zero for better S-expressions, raw fitness is the number of cases incorrectly handled.

As we saw in the second version of the block-stacking problem (where both efficiency and correctness were sought) and in the solution of differential equations (where both the solution curve and satisfaction of initial conditions were sought), the fitness function can incorporate both correctness and a secondary factor.

It is important that the fitness function return a spectrum of different values that differentiate the performance of individuals in the population. As an extreme example, a fitness function that returns only two values (say, a 1 for a solution and a 0 otherwise) provides insufficient information for guiding guide an adaptive process. Any solution that is discovered with such a fitness function is, then, essentially an accident. An inappropriate selection of the function set in relation to the number of environment cases for a given problem can create the same situation. For example, if the Boolean function OR is in the function set for the exclusive-or problem, this function alone satisfies three of the four environment cases. Since the initial random population of individuals will almost certainly numerous S-expressions equivalent to the OR function, we are effectively left with only two distinguishing levels of the fitness (i.e. 4 for a solution and 3 otherwise).

5.5. SELECTING THE PARAMETERS FOR THE RUNS

The fifth major step is the selection of the major and minor parameters of the algorithm and a decision on whether to use any of the four secondary genetic operations (described in section 7 below).

The selection of the population size is the most important choice. The population size must be chosen with the complexity of the problem in mind. In general, the larger the population, the better (Goldberg 1989). But, the improvement due to a larger population may not be proportional to the increased computational resources required. Some work has been done on the theory of how to optimally select the population size for string-based genetic algorithms (Goldberg l989); however, we can offer no corresponding theoretical basis for this tradeoff for hierarchical genetic algorithms at this time. Thus, selection of the population size lies in a category of external decisions that must be made by the user. In that respect, this decision is similar to the selection of the number of processing elements in neural nets, the selection of the string size for the condition parts of classifier system rules, and the selection of testing functions in ID3 type inductive systems. The problem of optimally allocating computer resources (particularly, population size and number of generations) over runs, the problem of optimally selecting other key parameters (such as percentage of individuals to participate in crossover and other genetic operations), and the problem of optimally parallelizing runs (e.g. cross migration versus independent isolated runs) are unsolved problems for all types of genetic algorithms.

5.6. TERMINATION AND SOLUTION IDENTIFICATION

Finally, the sixth major step is the selection of a termination criterion and solution identification procedure. The approach to termination depends on the problem. In many cases, the termination criterion may be implicitly selected by merely selecting a fixed number of generations for running the algorithm. For many problems, one can recognize a solution to the problem when one sees it (e.g. problems where the sum of differences becomes zero or acceptably close to zero). However, for some problems (such as time-optimal control strategy problems where no analytic solution is known), one cannot necessarily recognize a solution when one sees it (although one can recognize that the current result is better than any previous result or that the current solution is in the neighborhood of some estimate of the solution). The solution identification procedure used in this paper is to identify the best single individual of some generation where the termination criterion is satisfied as the solution to the problem (“winner takes all”).

There are numerous opportunities to use domain specific heuristic knowledge in connection with the genetic programming paradigm. Many of these areas have been studied in connection with string-based genetic algorithms (Grefenstette 1987b). First, it may be useful to include domain specific heuristic knowledge in creating the initial random population. For example, one might include sub-programs believed to be useful for solving the problem at hand in the initial random population. Or, one might use a probability distribution other than the uniform distribution to initially select the functions and terminals when the initial random individuals are randomly generated. Secondly, domain specific heuristic knowledge may be helpful in over-selecting or under-selecting certain points in the computer programs for the crossover operation. This may even include protecting certain points from selection for crossover under certain circumstances or requiring certain points to be selected for crossover under certain circumstances. Thirdly, domain specific heuristic knowledge may be useful in varying the parameters of the run based on information gained during the run. Fourth, domain specific heuristic knowledge can be used in the selection of the set of available functions and terminals for the problem so that this set is not merely minimally sufficient to solve the problem, but so that the set of available functions and terminals actively facilitates solution of the problem.

The extent to which one uses such domain specific heuristics is, of course, dependent on whether the primary objective is to solve a specific problem at hand or to study the process in the purest theoretical form. We have chosen not to use such domain specific heuristics in the work reported here.

6. COMPUTER IMPLEMENTATION

The author's computer program, consisting of 11562 lines of Common Lisp code, was primarily run on a Texas Instruments Explorer II+™ computer with a 40 megahertz LISP microprocessor chip with 32 megabytes of internal memory and a half gigabyte of external hard disk memory. Some runs were run on a microExplorer™ computer (an Apple Macintosh II™ computer with a Texas Instruments LISP expansion board) and some were run on a 4-node parallel Explorer MP™ computer.

6.1. COMPUTER INTERFACE

Robertson (1987) points out that an informative and interactive interface is an invaluable tool in carrying out computer experiments in the field of machine learning. Accordingly, the computer program used here has extensive interactivity, including eight full color graphs and five additional panes. There are various mouse-sensitive points for inspecting progress of a run while it is executing. The figure below shows this computer interface.

The first (upper left) graph dynamically tracks the average normalized fitness of the population. This graph also tracks the number of “hits” for the best single individual of each generation for problems where exact matches are possible (or the number of “near hits” for real-valued numerical problems). This number of “hits” or “near hits” is not used by the genetic algorithm in any way. The the number of “hits” or “near hits” are used only for monitoring the progress of runs and making the description of the run more understandable.

The second (upper right) graph dynamically tracks the average raw fitness of the population for each generation, the raw fitness of the best individual in the population, and the raw fitness of the worst individual in the population for each generation. This graph also displays the "baseline" average raw fitness of the initial random population as a dotted line.

The third graph is the “hits histogram” showing the number of individuals in the population with a particular number of “hits” (or “near hits”, for numerical problems) for a particular generation. This histogram a particularly informative view of the learning process. The horizontal axis of the hits histogram is the number of hits. The vertical axis of the hits histogram represents the number of individuals in the population with that number of hits. At the initial random generation, the bulk of the population appears at the far left of the histogram (with perhaps 0 or 1 hits). Then, after a few generations, the center of gravity of the hits histogram typically starts shifting from left to right. As learning takes place, this undulating “slinky” movement from left to right continues during the run. Finally, in the late stages of a run, individuals representing a perfect solution to the problem may start appearing at the far right of the histogram (which represents the maximum number of hits). Complete convergence occurs when 100% of the population becomes concentrated at the far right of the histogram. The possibility of premature convergence is suggested by a concentration of the population at one single sub-optimal number of hits in the histogram (and is verified by the fact that the individuals scoring that number of hits are identical). In contrast, normal progress towards a solution is typically indicated by a broad “flowing” distribution of individuals over many different numbers of hits in the histogram.

The fourth graph is a “fitness histogram” showing the number of individuals in the population having a fitness lying in a particular decile range of fitness values.

The fifth graph shows the best single S-expression of the current generation in both graphical and symbolic form. The internal and external points of the tree representing the S-expression are labeled with the appropriate functions and terminals. The sixth window shows the entire single best S-expression regardless of size.

The seventh window is used only on a subset of problems described in this paper, namely, the broom balancing problem. It shows the current state of the cart and broom.

The eighth graph is used only in a subset of the problems described in this paper to graphically display the single best individual in the population. The best S-expression typically changes with each generation. The vertical axis is the range of the target function. In the problems of sequence induction, symbolic function identification, symbolic regression, symbolic integration, symbolic differentiation, symbolic solution to differential equations, chaos, empirical discovery, and power series problems, this graph dynamically graphs the “target” function and the best individual S-expression from the current generation. In the symbolic integration and symbolic differentiation problems, the graph of the integral or derivative of the current best S-expression is added as an additional item. In the broom balancing problem, this graph is a three dimensional graph of the control surface.

The first (top) pane is a non-scrolling pane that provides general information about the run.

The second pane displays all the parameters of the run and may be updated with each individual S-expressions (and associated statistics) for the entire population for each generation (or selected generations) of the run. This window can be scrolled so that the progress of the run can be studied.

The third pane displays the number of hits achieved on the last generation of each previous run of the same problem. It is used for monitoring series of runs.

The fourth pane displays all the parameters of the run and is updated with two lines of information for each generation of the run. This information includes the single best individual S-expression for that generation, the raw fitness, adjusted fitness, normalized fitness, and number of hits for the best single individual in the population for that generation; the number of hits for the worst individual; and the average fitness of the population for that generation. This window can be scrolled so that the progress of the run can be studied.

The fifth pane is the LISP command pane.

6.2. SPECIAL PROBLEMS OF COMPUTER IMPLEMENTATION

In order to replicate the results reported herein or to further explore the potential of the "genetic programming" paradigm described in this article, it is first necessary to implement the algorithm on a computer. Because the paradigm described herein involves executing and modifying computer programs in non-standard ways and because these computer programs were either originally generated at random or created genetically, a number of practical computer implementation issues come to the forefront. Some of the techniques required are not described in the available documentation of the relevant programming languages or computing machines and some do not work as one might reasonably expect. In this section, we discuss some of the many pitfalls that abound in this area.

First, it should be noted that if the experimenter chooses to use the Common LISP function EVAL to implement the measurement of fitness of individual LISP S-expressions, the evaluation will work correctly only if all of the variable atoms appearing in the given S-expressions are global variables.

Secondly, the most efficient implementation of the crossover operation known to the author uses the COPY-TREE and RPLACA functions in LISP. First, the COPY-TREE function is used to make a copy of each parent. Then, the RPLACA function is used to destructively change the pointer of the CONS cell of the copy of one parent at its crossover point so that it points to the crossover fragment (subtree) of the copy of the other parent. Then, the RPLACA function is used to destructively change the pointer of the CONS cell of the copy of second parent at its crossover point so that it points to the crossover fragment (subtree) of the copy of the first parent. After destructively changing the pointers in the copies, the resulting altered copies become the offspring. The original parents remain in the population and can often repeatedly participate in other operations during the current generation. That is, the selection of parents is done with replacement (i.e. reselection) allowed.

Third, because the genetic programming paradigm described herein involves executing randomly generated computer programs, the individuals in the initial random population as well as the individuals produced in later generations of the process often have sub-expressions which evaluate to astronomically large numbers or very small numbers. When the range is integral, the BIGNUM mode is automatically used in the Common LISP programming language. In this mode, integer numbers can grow arbitrarily large (limited only by the virtual address space of the machine). Thus, the potential growth in size of the integers produced by the randomly generated S-expressions presents no problem, as a practical matter. On the other hand, when the range is real-valued, floating point overflows or underflows will frequently occur. In problems involving such floating point variables, it is therefore a practical necessity to wrap the entire algorithm in error handlers that accommodate every possible kind of floating point underflow and overflow applicable to the particular computer involved.

Fourth, it is important to note that this hierarchical genetic algorithm (as with genetic algorithms in general) is probabilistic in the following four different ways: (a) the initial population is typically generated entirely at random from the available functions and terminals; (b) both parental individuals participating in the crossover operation are chosen at random; (c) the crossover points within each parent are selected at random (using a probability distribution); and (d) the individuals undergoing the operation of fitness proportionate reproduction are chosen randomly in proportion to normalized fitness. Thus, in implementing genetic algorithms on a computer, it is important to have an effective randomizer that is capable of producing the stream of seemingly independent random integers needed by the algorithm. Many randomizers originally written for the purpose of generating random floating point numbers are not suitable for this purpose. A randomizer with 3 independent seeds was used here. It is also important to have the option of seeding the randomizer so that interesting runs can potentially be replicated (e.g. perhaps with additional details displayed, such as the audit trail mentioned above).

Fifth, in problems involving iteration with a DU ("Do-Until") operator, a DUL operator, or a SIGMA summation operator, it is necessary to suppress premature evaluation of the WORK and PREDICATE arguments (if any) of the operator. Neither the WORK argument nor the PREDICATE argument (if any) of such operators are to be evaluated outside the operator. Instead, these argument(s) must be evaluated dynamically inside the operator on each iteration. Because of the evaluation model of Common LISP, such operators cannot be implemented directly as functions in Common LISP. The reason is that the argument(s) would be evaluated prior to entry into the function and the operator would, for example, then merely repeatedly evaluate the value of the WORK as opposed to doing the WORK itself.

This problem can be solved by implementing such iterative operators as a Common LISP macro with a related function. For example, the iterative DU operator should expand into, for example, `` `(DU-1 `,WORK `,PREDICATE) `` (Rice 1989). Then the body of the related function DU-1 is implemented so as to have the desired iterative behavior (i.e. `(LOOP DO (EVAL WORK) UNTIL (EVAL PREDICATE)))`. This is possible because the arguments to the sub-expressions WORK and PREDICATE are bound in the global environment. We could, in principle, have addressed this problem by introducing a quoting operator into the set of functions so as to allow DU to have the semantics of DU-1. But this approach results in incorrect performance whenever the QUOTE function happens to occur at a crossover point and becomes separated from its intended argument. Moreover, some implementations of Common LISP (the Texas Instruments Explorer machines being among this group) use a technique called "macro displacement" to side-effect programs being interpreted with the macro-expanded version. This has the beneficial effect of speeding up execution by incurring the cost of the macro-expansion only once. However, because this technique side-effects the program itself, if macro displacement is not disabled for genetic operators, then crossover that occurs on individuals after macro-expansion may see forms that are introduced by the macro-expander, not forms that are really part of the problem. On Texas Instruments machines, this behavior can be disabled by setting "si:inhibit-displacing-flag" to T.

Sixth, when iterative operators (such as DU, DUL, and SIGMA) are used, individual S-expressions in the population will often contain an unsatisfiable termination predicates. Thus, it is a practical necessity (when working on a serial computer) to place limits on both the number of iterations allowed by any one execution of a such an operator. Moreover, since the individuals S-expressions in the genetic population often contain deep nestings of such operators, a similar limit must be placed on the total number of iterations allowed for all such operators that may be evaluated in the process of evaluating any one individual S-expression for any particular environmental case. Thus, the termination predicate of each operator is actually an implicit disjunction of the explicit predicate for that operator and two additional implicit termination predicates. The typical "time out" limits that we have used are that the DU operator "times out" if there have been more than 25 iterations for an evaluation of a single DU operator or if there have been a total of more than 100 iterations for all DU operators that are evaluated for a particular individual S-expression for a particular environmental case. Of course, if we could execute all the individual LISP S-expressions in parallel (as nature does) so that the infeasibility of one individual in the population does not bring the entire process to a halt, we would not need these limits. Note, however, that even when a DU operator times out, it nevertheless returns a value. In particular, the DU operator evaluates to T unless one of the two implicit termination predicates times out. The value resulting from this evaluation of the DU operator is, of course, in addition to the side effects of the DU function on the state variables of the system (particularly the STACK and TABLE in the block-stacking problem). If the predicate of a DU operator is satisfied when the operator is first called, then the DU operator does no work at all and simply returns a T. The DUL operator and the SIGMA operator are treated similarly.

Seventh, for all but the simplest problems discussed in this paper, the overwhelming majority of computer time is consumed by the evaluation of fitness of the individuals (rather than, as one might suppose, the actual genetic operations or other administrative aspects of the program). This is also often true for genetic algorithm problems in general. When the fitness calculation consumes the overwhelming majority of computer time, then fine-grained parallelism (as opposed to coarse-grained parallelism) and the techniques of data parallelism confer no particular advantage. Similarly, if this concentration exists, ones efforts at optimization must necessarily be focused almost entirely on the relatively small number of lines of code that are used to compute fitness (over the various environmental cases of the particular problem). One highly effective way to optimize the fitness calculation is to create a look-up table of S-expressions that have been previously encountered so that their fitness need not be recomputed. This hash table can span both generations and runs (provided the environmental cases remain the same). Note that the technique of look-up tables is, however, inconsistent with the technique of changing the environmental cases on every generation so as to minimize the possible bias arising from the necessarily small sampling of environmental cases used on problems where fitness computations are time-consuming.

Eighth, many problems involve time-consuming transcendental functions (e.g. EXP, SIN, COS) that are computed via Taylor power series. In some problems, both the initial randomly-generated individuals and the later genetically-created individuals in the population often contain multiple occurrences of these functions within a single individual. On other problems, these functions appear in the state transition equations of the problem, rather than the individual S-expressions. In both cases, a considerable amount of computer time can be saved by evaluating these functions via table look-up, rather than direct computation.

6.3. MINOR PARAMETERS FOR CONTROLLING THE ALGORITHM

Five minor parameters are used to control the process. Two of them control the frequency of performing the genetic operations; one of them controls the percentage of internal (function) points chosen as crossover points; and two of them help conserve computer time.

The same values of these five minor parameters were used in all problems described in this article.

First, crossover was performed on 90% of the population for each generation. That is, if the population size is 300, then 270 individuals (135 pairs) from each generation were selected (with reselection allowed) from the population with a probability equal to their normalized fitness.

Second, fitness proportionate reproduction was performed on 10% of the population on each generation. That is, 30 individuals from each generation were selected (with reselection allowed) from the population with a probability equal to their normalized fitness.

Third, we used a non-uniform probability distribution over the potential crossover points in the parents. It allocates 90% of the crossover points equally amongst the internal (function) points of each tree. It also allocates the remainder (i.e. 10%) of the crossover points are allocated equally amongst the external (terminal) points of each tree. We believe this non-uniform probability distribution promotes the recombining of larger structures than would be the case with a uniform distribution over all points (which may do an inordinate amount of mere swapping of terminals from tree to tree in a manner more akin to point mutation as opposed to true recombining of "building block" substructures).

Fourth, a maximum depth of 15 was established for S-expressions created by the crossover operation. This limit prevents large amounts of computer time being expended on a few extremely deep (and usually highly unfit) individual S-expressions. Of course, if we could execute all the individual LISP S-expressions in parallel (as nature does) in a manner such that the infeasibility of one individual in the population does not disproportionately jeopardize the resources needed by the population as a whole, we would not need this kind of artificial limit. If a crossover between two parents would create an individual whose depth would exceed this limit, the contemplated crossover operation is simply aborted and replaced with an instance of fitness proportionate reproduction for the two parents. This limit is typically reached only on a few crossovers per run (if any).

Fifth, a maximum depth of 4 was established for the random individuals generated for the initial random generation (generation 0).

The first three of these minor parameters are probably acceptable for most problems that might be encountered. It might be advisable to increase the last two of these minor parameters on problems where the structure of the solution is likely to be highly complex.

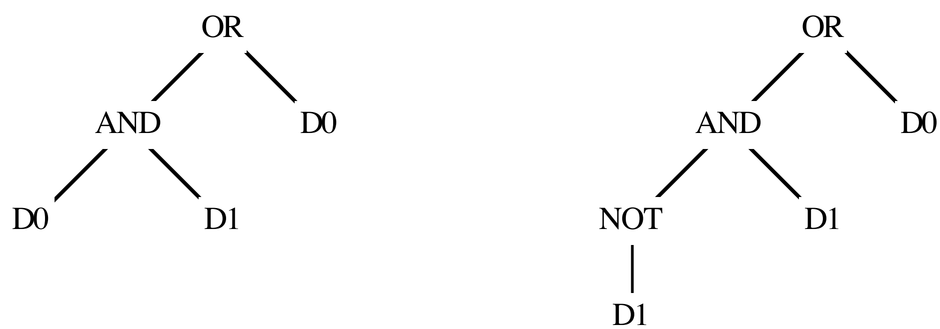
7. ADDITIONAL OPERATIONS

In addition to the two primary genetic operations of fitness proportionate reproduction and crossover, there are four secondary operations for modifying the structures undergoing adaptation. They are mutation, permutation, editing, and the "define building block" operation.

7.1. THE MUTATION OPERATION

The mutation operation provides a means for introducing small random mutations into the population.

The mutation operation is an asexual operation in that it operates on only one parental S-expression. The individual is selected proportional to normalized fitness. The result of this operation is one offspring S-expression. The mutation operation selects a point of the LISP S-expression at random. The point can be an internal (function) or external (terminal) point of the tree. This operation removes whatever is currently at the selected point and inserts a randomly generated sub-tree at the randomly selected point of a given tree. This operation is controlled by a parameter which specifies the maximum depth for the newly created and inserted sub-tree. A special case of this operation involves inserting only a single terminal (i.e. a sub-tree of depth 0) at a randomly selected point of the tree. For example, in the figure below, the third point of the S-expression shown on the left below was selected as the mutation point and the sub-expression (NOT D1) was randomly generated and inserted at that point to produce the S-expression shown on the right below.



Before After

The mutation operation potentially can be beneficial in reintroducing diversity in a population that may be tending to prematurely converge. Our experience has been that no run using only mutation and fitness proportionate reproduction (i.e. no crossover) ever produced a solution to any problem (although such solutions are theoretically possible given enough time). In other words, "mutating and saving the best" does not work any better for hierarchical genetic algorithms than it does for string-based genetic algorithms. This negative conclusion as to the relative unimportance of the mutation operation is similar to the conclusions reached by most research work on string-based genetic algorithms (Holland 1975, Goldberg 1989).

7.2. THE PERMUTATION OPERATION

The permutation operation is both an extension of the inversion operation for string-based genetic algorithms to the domain of hierarchical genetic algorithms and a generalization of the inversion operation.

The permutation operation is an asexual operation in that it operates on only one parental S-expression. The individual is selected in a manner proportional to normalized fitness. The result of this operation is one offspring S-expression. The permutation operation selects a function (internal) point of the LISP S-expression at random. If the function at the selected point has k arguments, a random permutation is selected at random from the set of k! possible permutations. Then the arguments of the function at the selected point are permuted in accordance with the random permutation. Notice that if the function at the selected point happens to be commutative, there is no immediate effect from the permutation operation on the value returned by the S-expression.

The inversion operation for strings reorders the order of characters found between two selected points of a single individual by reversing the order of the characters between the two selected points. The operation described here allows any one of k! possible permutations to occur (of which the reversal is but one).

The permutation operation can potentially bring closer together elements of a relatively high fitness individual so that they are less subject to later disruption due to crossover. However, like the mutation operation, our experience, after including the permutation operation in numerous runs of various problems described herein, is that the benefits of this operation are purely potential and have yet to be observed.

7.3. THE EDITING OPERATION

The editing operation provides a means to edit (and simplify) S-expressions as the algorithm is running.

The editing operation is applied after the new population is created through the action of the other operations. The editing operation is an asexual operation in that it operates on only one parental S-expression. The result of this operation is one offspring S-expression. All of the previously described operations operate on individuals selected in proportion to fitness. The editing operation is the exception. The editing operation, if it is used at all, is applied to every individual S-expression in the population.

The editing operation recursively applies a pre-established set of editing rules to each S-expression in the population. First, in all problem domains, if any sub-expression has only constant atoms as arguments, the editing operation will evaluate that sub-expression and replace it with the value obtained. In addition, the editing operation applies particular sets of rules that apply to various problem domains, including rules for numeric domains, rules for Boolean domains, etc. In numeric problem domains, for example, the set of editing rules includes rules that insert zero whenever a sub-expression is subtracted from an identical sub-expression and also includes a rule that inserts a zero whenever a sub-expression is multiplied by zero. In Boolean problem domains, the set of editing rules includes a rule that inserts X in place of `(AND X X)`, `(OR X X)`, or `(NOT (NOT X))`, etc. The editing operation is controlled by a frequency parameter which specifies whether it is applied on every generation or merely a certain number of the generations.

The main reason for the editing operation is convenience. It simplifies S-expressions and saves computer resources. It also appears to improve overall performance slightly. The editing operation apparently improves performance by reducing the vulnerability of an S-expression to disruption due to crossover at points within a potentially collapsible, non-parsimonious sub-expression. Crossover at such points typically leads to counter-productive results. For example, consider the sub-expression `(NOT (NOT X))`. This sub-expression could be simplified to the more parsimonious sub-expression X. In this example, a crossover in the middle of this sub-expression would usually produce exactly the opposite Boolean result as the expression as a whole. In this example, the editing operation would prevent that kind of crossover from occurring by condensing the sub-expression to the single term X.

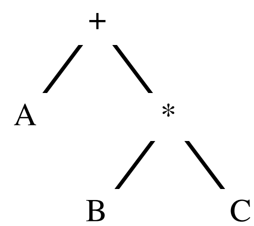
7.4. THE "DEFINE BUILDING BLOCK" OPERATION

The "define building block" operation is a means for automatically identifying potentially useful "building blocks" while the algorithm is running.

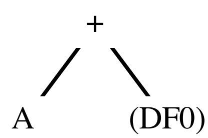
The "define building block" operation is an asexual operation in that it operates on only one parental S-expression. The individual is selected proportional to normalized fitness. The operation selects a function (internal) point of the LISP S-expression at random. The result of this operation is one offspring S-expression and one new definition. The “define building block” operation works by defining a new function and by replacing the sub-tree located at the chosen point by a call to the newly defined function. The newly defined function has no arguments. The body of the newly defined function is the sub-tree located at the chosen point. The newly defined functions are named DF0, DF1, DF2, DF3, ... as they are created.

For the first occasion when a new function is defined on a given run, “(DF0)” is inserted at the point selected in the LISP S-expression. The newly defined function is then compiled. The function set of the problem is then augmented to include the new function. Thus, if mutation is being used, an arbitrary new sub-tree grown at the selected point has the potential to include the newly defined function.

For example, consider the simple LISP S-expression `(+ A (* B C))` shown, in graphical form, below:



Suppose that the third point (i.e. the multiplication) is selected as the point for applying the "define building block" operation. Then, the subtree for `(* B C)` is replaced by a call to the new "defined function" DFO producing the new S-expression `(+ A (DF0))` shown, in graphical form, below:



This new tree has the call `(DF0)` in lieu of the sub-tree `(* B C)`.

At the same time, the function DFO was created. If this new "defined function" were written in LISP, it would be written as shown below:

```
(defun DF0 ()
  (* B C)
)
```

In implementing this operation on the computer, the sub-tree calling for the multiplication of B and C is first defined and then compiled during the execution of the overall run. The LISP programming language facilitates this "define building block" operation in two ways. First, the form of data and program are the same in LISP and therefore a program can be altered by merely performing operations on it as if it were data. Secondly, it is possible to compile a new function during the execution of an overall run and then execute it.

The effect of this replacement is that the selected sub-tree is no longer subject to the potentially disruptive effects of crossover because it is now an indivisible single point. In effect, the newly defined indivisible function is a potential “building block” for future generations and may proliferate in the population in later generations based on fitness.

Note that the original parent S-expression is unchanged by the operation. Moreover, since the selection of the parental S-expression is in proportion to fitness, the original unaltered parental S-expression may participate in additional genetic operations during the current generation, including replication (fitness proportionate reproduction), crossover (recombination), or even another “define building block” operation.

8. ROBUSTNESS

The existence and nurturing of a population of disjunctive alternative solutions to a problem allows the genetic programming paradigm to perform effectively even when the environment changes. To demonstrate this, the environment for generations 0 through 9 is the quadratic polynomial $x^2 + x + 2$; however, at generation 10, the environment abruptly changes to the cubic polynomial $x^3 + x^2 + 2x + 1$; and, at generation 20, it changes again to a new quadratic polynomial $x^2 + 2x + 1$. A perfect-scoring quadratic polynomial for the first environment was created by generation 3. Normalized average population fitness stabilized in the neighborhood 0.5 for generations 3 through 9 (with genetic diversity maintained). Predictably, the fitness level abruptly dropped to virtually 0 for generation 10 and 11 when the environment changed. Nonetheless, fitness increased for generation 12 and stabilized in the neighborhood of 0.7 for generations 13 to 19 (after creation of a perfect-scoring cubic polynomial). The fitness level again abruptly dropped to virtually 0 for generation 20 when the environment again changed. However, by generation 22, a fitness level again stabilized in the neighborhood of 0.7 after creation of a new perfect-scoring quadratic polynomial.

9. PERFORMANCE MEASUREMENT

It is difficult to meaningfully compare performance among disparate machine learning paradigms.

9.1. FACTORS INVOLVED COMPARING MACHINE LEARNING PARADIGMS

A comparison of machine learning paradigms involves many different factors. Among these factors are the following:

- the machine resources required to execute the algorithm (notably computer time and computer memory),
- the generality of the algorithm in terms of the range of problems that can be solved,
- the qualitative features of the output (e.g. presence of default hierarchies),
- the quantitative features of the output of the algorithm (e.g. parsimony),
- the ability of the algorithm to scale up,
- the extent and character of preprocessing of the inputs that is required,
- the intelligibility of the solution in terms of whether it is easily understood in terms appropriate to the problem domain involved,
- the amount of information about the size and shape of the ultimate solution that is explicitly (or implicitly) provided by the user,
- the ability to provide an audit trail showing how the algorithms arrived at its solution to a problem,
- the ability to verify the validity of the output,
- the ability to incrementally modify a solution and, in particular, the ability to robustly (i.e. rapidly and effectively) modify one solution to handle slightly different situations,
- the ability of the algorithm to yield near linear speed-ups in highly parallel computer architectures (and thereby solve very large problems),
- the ability of the algorithm to be fault-tolerant in the sense that the algorithm can operate reasonably well even if some of the processors or data are lost at some point during the run,
- the accuracy of the paradigm, and
- the probabilistic or deterministic nature of the algorithm.

We do not attempt to make a thorough comparison of the various machine learning paradigms here. Even comparing a seemingly quantitative factor as computer time requires selecting benchmark problems from an infinity of potential problems. Some paradigms are especially well suited to solving certain types of problems. More importantly, some paradigms simply cannot solve certain types of problems at all. Nothing is as slow as <u>never</u> in solving a particular problem! Limiting the test suite of problems to the lowest common denominator of problems solvable by all existing machine learning paradigms merely favors the most specialized paradigms with the least problem-solving power. On the other hand, broadening the test suite, in effect, automatically negatively prejudges all paradigms which cannot solve (or are not well suited to solving) the added problems.

Even comparing a seemingly quantitative factor as computer time requires selecting a particular computer on which to make the benchmark comparison. Various computers may be more or less well suited for executing certain paradigms. For example, a particular parallel computer architecture may be advantageous or disadvantageous to certain paradigms; a machine with a modest amount of memory may be advantageous or disadvantageous to certain paradigms; a LISP workstation would be especially well-suited to a LISP paradigm while other nominally fast workstations often require vast amounts of time to perform relatively simple LISP operations; etc.

The Boolean 6-multiplexer problem offers an opportunity to compare speed for a very narrow, but interesting, class of problems that has been studied in the literature in connection with three other machine learning paradigms, namely, neural nets, classifier systems, and inductive decision trees. Wilson (1987) reported about an order of magnitude fewer iterations than a neural net used by Barto et. al. (1985) for this particular problem (about 12,000 to 15,000 iterations, as compared with almost 200,000. Somewhat different iterative steps were involved in the two paradigms although both types of iterative steps corresponded to the number of examples seen by the system. Quinlan (1988) reported even faster performance than Wilson's using inductive decision trees (using the C4 paradigm). The processing of individuals in genetic programming paradigm uses considerably more computational resources (measured via time and memory) than Wilson's or Quinlan's approach to this particular problem. The genetic programming paradigm has found solutions to the Boolean 6-multiplexer problem after processing as few as 2400 individuals (8 generations of 300). It usually produces solutions after processing 9,000 individuals (30 generation of 300). If number of examples seen by the system is considered, this would be higher (but in the same neighborhood) as Barto's approach. But, as discussed below, a requirement of processing 511,500 individuals can be considered as one reasonable measure of the processing requirements for the genetic programming paradigm for this particular problem when the effect of unsuccessful runs is also considered.

As to the range of problems that can be solved, the genetic programming paradigm described herein can solve a number of types of problems that cannot be solved (or, at least, have not been solved) with existing machine learning paradigms. The fact that the solutions come in the form of computer programs (which can contain a mixture of conditional logical operations, mathematical functions, domain specific operations and functions, and recursions and iterations) widens the range of machine learning problems that can be easily handled (or handled at all). The variety of different problems discussed in this paper suggests the wide variety of problems susceptible to the genetic programming paradigm.

As to qualitative features of the output of algorithm, the algorithm described herein is inherently hierarchical.

In addition, the genetic programming paradigm often produces results expressed in terms of default hierarchies.

As to quantitative features of the output of the algorithm, the inherently hierarchical character of the output also contributes to the parsimony of the results.

A key issue in artificial intelligence and machine learning is how to "scale up" promising techniques that succeed on small sub-problems to larger problems. One way to solve large problems is, in general, to decompose the large problem hierarchically into smaller sub-problems ("building blocks"). It is plausible that having hierarchical results for smaller problems would be a precondition to scaling up to larger problems. The inherently hierarchical character of the results, in turn, should lead to relatively high ability to scale up, although this has not been specifically demonstrated here. Moreover, the "define building blocks" operation appears to facilitate the identification of useful "building blocks" with which to construct hierarchical solutions to problems.

As to the amount and type of preprocessing required, the algorithm used here requires no preprocessing of inputs. The inputs are presented directly in the terms of the observed state variables of the problem domain involved. This lack of preprocessing is a major distinction relative to typical neural network and other machine learning algorithms.

As to intelligibility, the absence of preprocessing, the direct use of functions that come from the problem domain, the inherently hierarchical character of the algorithm, and the editing operation contributes to the intelligibility of the results in terms of the problem domain involved.

As to requiring advance information about the size and shape of the ultimate solution, the genetic programming paradigm uses relatively little such advance knowledge. This is in contrast with the pre-specifying of the precise string size (6) in the classifier system approach to the Boolean multiplexer problem, the pre-specifying of the maximum number of states for the finite automaton (32) in the string-based approach to the “artificial ant” problem, and the pre-specifying of the precise number of processing units at each layer (and number of layers) required by most neural network paradigms.

As to auditability, genetic algorithms in general allow construction of an audit trail showing how the algorithm arrived at its solution to a problem.

As to ability to verify the results, the fact that there is no preprocessing of inputs and the final result is expressed directly in terms of functions that are relevant to the problem domain make sit relatively easy to verify solutions generated by the algorithm.

As to ability to incrementally and robustly (i.e. rapidly and effectively) modify one solution to handle different situations, genetic algorithms have this ability (as contrasted with, say, the neural net back propagation paradigm).

As to ability to profitably parallelize the algorithm, genetic algorithms in general yield near linear speed ups when executed on parallel computer architectures.

As to fault-tolerance, genetic algorithms in general are fault-tolerant in the sense that the algorithm continue to operate well even if some of the population is lost at some point during the run.

As to accuracy, genetic algorithms in general are reasonably accurate in arriving at solutions based on the guidance provided by the fitness function.

As to determinism, genetic algorithms in general are probabilistic and not deterministic algorithms. Particular interesting runs can be re-created by seeding the randomizer.

Although we do not attempt to make a thorough comparison of the various machine learning paradigms here, we can, nonetheless, perform the limited task of determining how much processing of individuals is required to solve some of the problems discussed herein.

9.2. PREMATURE CONVERGENCE AND NICHES

Genetic algorithms, in general, do not always produce results on a particular run of the algorithm. For one thing, genetic algorithms inherently involve probabilistic steps at four major points (cited earlier) in the algorithm. These probabilistic steps can be viewed as "initial conditions" of uncontrolled chaotic variables. Because of these probabilistic steps, some runs simply do not produce desired results. In addition, genetic algorithms sometimes simply fail to converge at all on a given run within a particular amount of time. More commonly, genetic algorithms may prematurely converge (i.e. converge to a sub-optimal result). The exponentially increasing allocation of future trials on the basis of the currently observed fitness statistics of the population is both the strength and a weakness of genetic algorithms. This allocation is a weakness because it may result in premature convergence (Booker 1987). This allocation is a strength because it is the fundamental reason why genetic algorithms work in the first place.

We believe that premature convergence (or occasional non-convergence for other reasons) should be viewed as an inherent aspect of genetic algorithms (rather than as a problem to be "cured" by trying to alter the fundamental nature of the algorithm). Nature can be the guide here. Premature convergence manifests itself in nature in the form of the niche preemption principle. That is, in nature, biological niches tend to be dominated by a single species (Maguarran 1988). Premature convergence in a particular niche (and occasional complete failure in a particular niche) is the norm in nature. Thus, each particular run of a genetic algorithm can be viewed as a niche, and, particular runs of a genetic algorithm can and will become dominated by particular sub-optimal species.

Nature carries out its genetic experiments in parallel in many niches at a time. Thus, nature avoids the effects of randomness, niche preemption, premature convergence, chaotic affects, and unfortuitous initial conditions by conducting her experiments many times in parallel in many (often functionally equivalent) niches. The best individual from these multiple experiments in separate, isolated niches is then potentially available to proliferate in the future.

Similarly, the effects of randomness, premature convergence, unfortuitous initial conditions, and other chaotic affects on genetic algorithms can be minimized by making entirely separate multiple independent runs. (These multiple independent runs, of course, lend themselves to parallel computer architectures and yield almost complete linear speed-up, but that is not the point). Note that the multiple independent runs we are talking about here do not involve cross-migration (cf. Tanese 1989). The best single individual from all of these multiple independent runs is then identified as the solution to the problem.

9.3. AMOUNT OF PROCESSING REQUIRED

We can undertake to measure genetic algorithms in terms of the likely number of niches (i.e. multiple independent runs) needed to produce a desired result with a certain probability, say, $z = 99\%$.

If, for example, a particular run of a genetic algorithm produces the desired result with only a probability of success p_s after a specified number of generation N_{gen} with a population of size N, then the probability of achieving the desired result at least once within K runs is $1 - (1-p_s)^K$. If we are seeking to achieve the desired result with a probability of, say, $z = 1 - \varepsilon = 99\%$, then the number K of independent runs (niches) required is $K = \log(1-z) / \log(1-p_s) = \log \varepsilon / \log(1-p_s)$, where $\varepsilon = 1-z$.

The number of independent runs (niches) K obtained from this formula can then be multiplied by the amount of processing required for the run. The amount of processing required is generally proportional to the product of the population size and the number of generations executed.

For example, we ran 248 runs of the Boolean 6-multiplexer with a population size of 300 and found that the probability of success p_s (of finding one individual S-expression that produced the correct Boolean output for all 64 combinations of the 6 Boolean inputs) was 23% after 51 generations. With a probability of success p_s of 23%, 18 independent runs are required to assure a 99% probability of solving the problem.

We then tried a population of 500 and found that the probability of success p_s was 68% with a population of 500 after 51 generations. With a probability of success p_s of 68%, only 4 independent runs are required to assure a 99% probability of solving the problem. Even after the 4 runs are adjusted for the 1.67 times greater computing effort required to process 500 individuals, instead of 300, the population size of 500 produces the desired solution with only about one third the computational effort of the population size of 300. We noted a similar advantage to the population size of 500 for several other problems, including the artificial ant problem. Therefore, for the purposes of presenting performance statistics in this section, we have used a population size of 500 throughout this section.

In this section, we determine the number of probability of success p_s for a test suite of twelve problems:

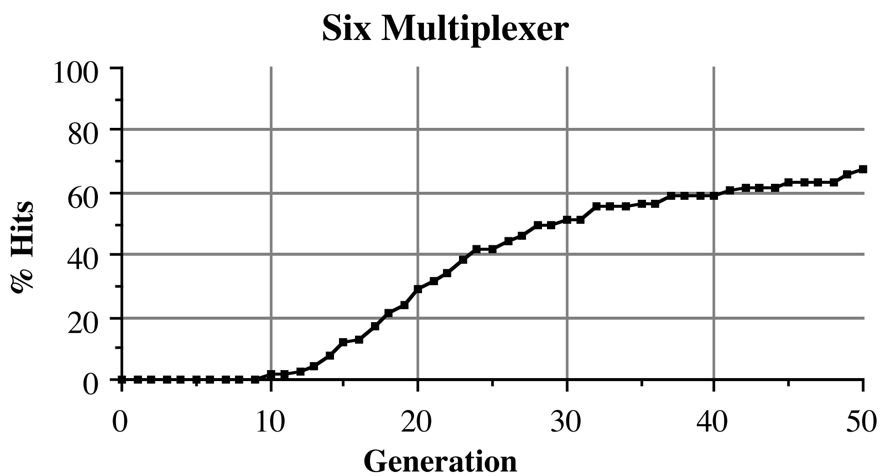
- the Boolean 6-multiplexer,
- block-stacking,
- artificial ant (with the John Muir trail),
- sequence induction (Fibonacci sequence),
- sequence induction (Hofstadter sequence),
- symbolic regression (with $2.718x^2+3.1416x$ as the target function),

- symbolic "data to function" integration (with $4x^3 + 3x^2 + 2x + 1$ as the function to be integrated and $x^4 + x^3 + x^2 + x$ as the desired result),
- symbolic "data to function" differentiation (with Sin $x + x^2 + x$ as the function to be differentiated and Cos $x + 2x + 1$ as the desired result),
- differential equations ($dy/dx - 2y + 4x = 0$),
- automatic programming (solving a pair of linear equations),
- the pursuer-evader differential game, and
- symbolic multiple regression (with the complex multiplication function as the target function).

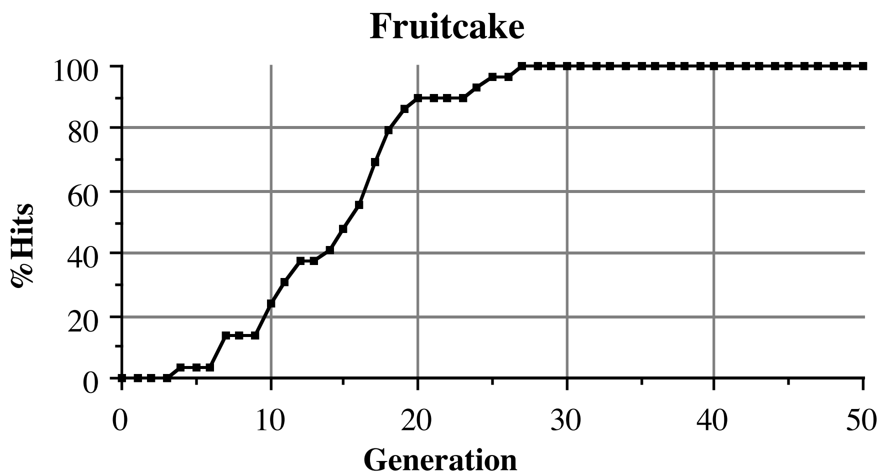
We began running between 25 and 200 runs (usually 100) of each of the problems in the test suite. For each problem, we used a population size of 500 for each run. For each problem, we ran 51 generations for each run. For each problem, we noted the probability of success p_s that of finding the desired solution by generation 0, 1, 2, ..., 50.

Note that the total number of individuals that need to be processed is a function of the percentage (z = 99%) and everything that affects the probability p_s. The probability of success p_s is, in turn, a function of (1) the two primary parameters of the algorithm (i.e. the population size 500 and the number of generations 51), (2) the five fixed secondary parameters of the genetic algorithm (described in section 6.3), and (3) all the fixed minor details of our implementation of the algorithm.

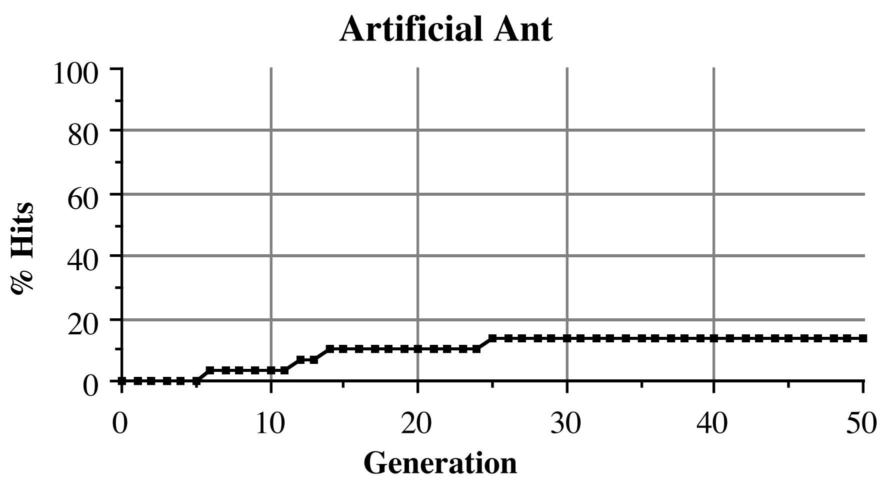
For the 6-multiplexer problem, the probability of success p_s that at least one individual S-expression produces the desired Boolean output for all 64 combinations of the 6 Boolean inputs is displayed on the graph below as a function of the number of generations. In particular, the probability of success p_s after 50 generations is 67%.



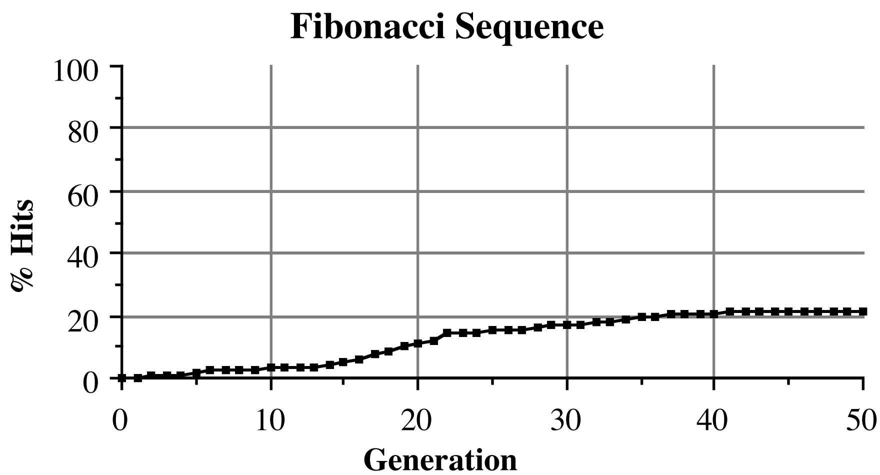
For the block-stacking problem, the probability of success p_s that at least one individual plan (S-expression) restacks the blocks so that they are all on the STACK spelling "FRUITCAKE" (within both iterative "time out" limits) in all 166 environmental starting condition cases is displayed on the graph below as a function of the number of generations. In particular, the probability of success p_s after 50 generations is 100%. In fact, this 100% level is achieved at generation 27.



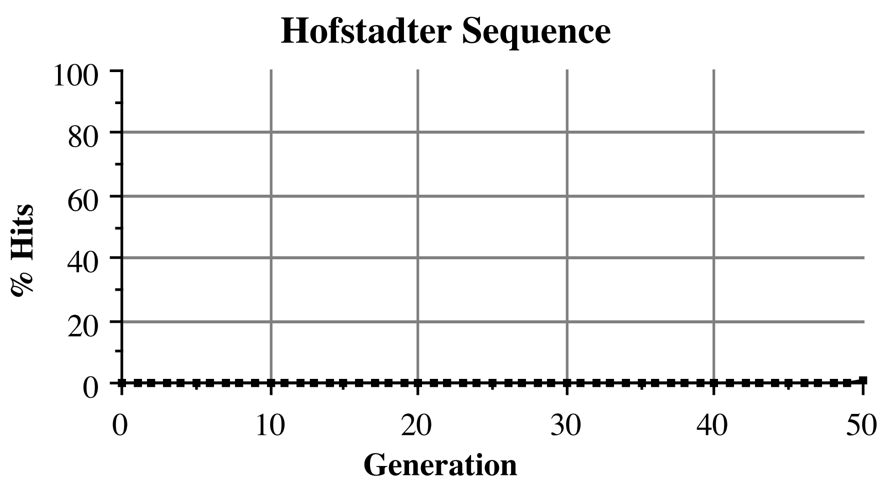
For the artificial ant problem (with the John Muir trail), the probability of success p_s that at least one individual plan (S-expression) causes the ant to traverse the entire trail and collect all 89 stones (within the overall "time out" limit) is displayed on the graph below as a function of the number of generations. In particular, the probability of success p_s after 50 generations is 14%.



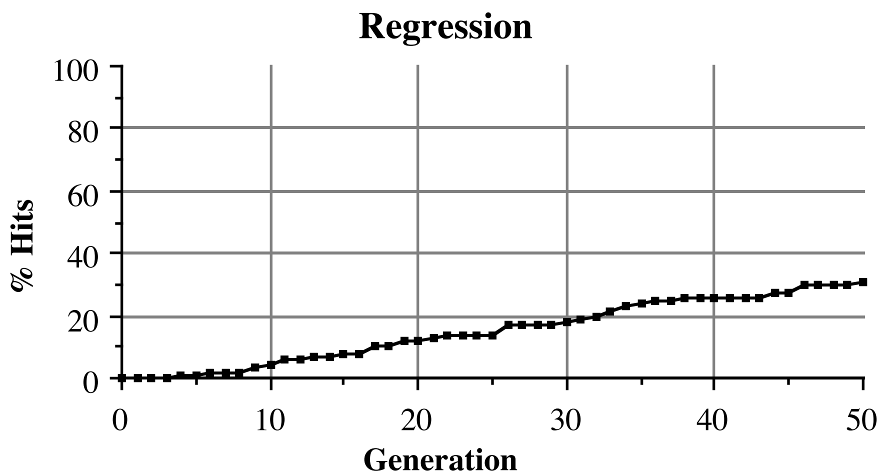
For the sequence induction problem involving the Fibonacci sequence, the probability of success p_s that at least one individual S-expression produces the correct integer value of the sequence for the first 13 non-base positions of the sequence is displayed on the graph below as a function of the number of generations. In particular, the probability of success p_s after 50 generations is 22%.



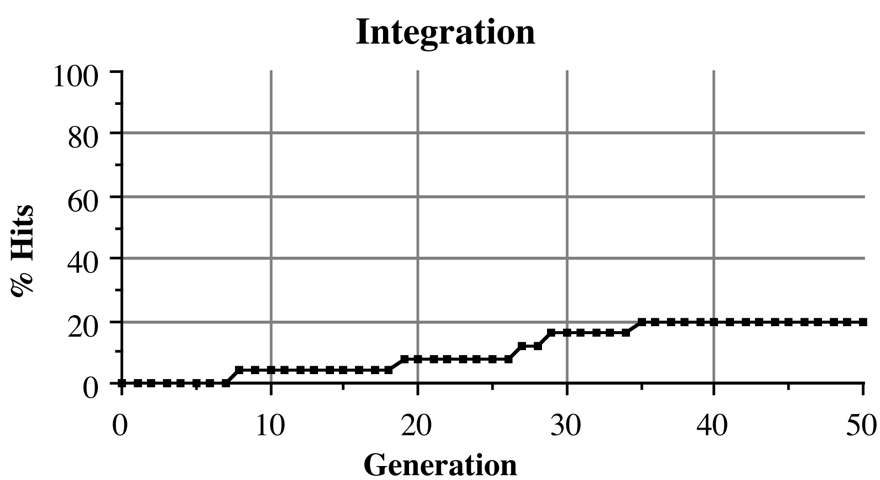
For the sequence induction problem involving the Hofstadter sequence, the probability of success p_s that at least one individual S-expression produces the correct integer value of the sequence for the first 13 non-base positions of the sequence is displayed on the graph below as a function of the number of generations. In particular, the probability of success p_s after 50 generations is 1%.



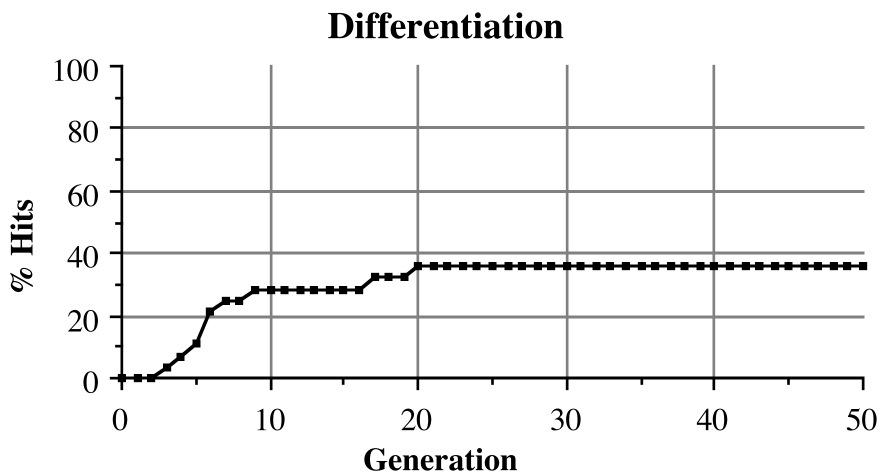
For the symbolic regression problem (with $2.718x^2+3.1416x$ as the target function), the probability of success p_s that at least one individual S-expression comes within 1% of the target function for all 20 random values of the independent variable is displayed on the graph below as a function of the number of generations. In particular, the probability of success p_s after 50 generations is 31%.



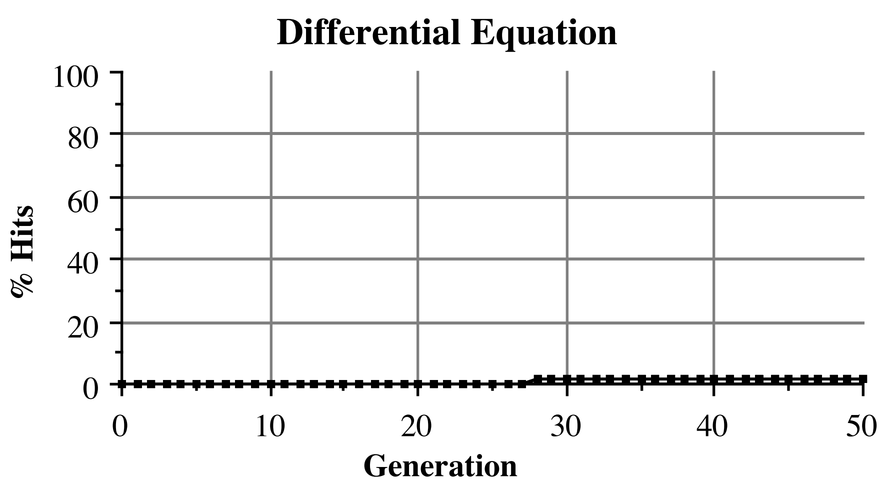
For the symbolic "data to function" integration problem (with $4x^3 + 3x^2 + 2x + 1$ as the function to be integrated and $x^4 + x^3 + x^2 + x$ as the desired result), the probability of success p_s that at least one individual S-expression comes within 1% of the integral of target function for all 50 random points is displayed on the graph below as a function of the number of generations. In particular, the probability of success p_s after 50 generations is 20%.



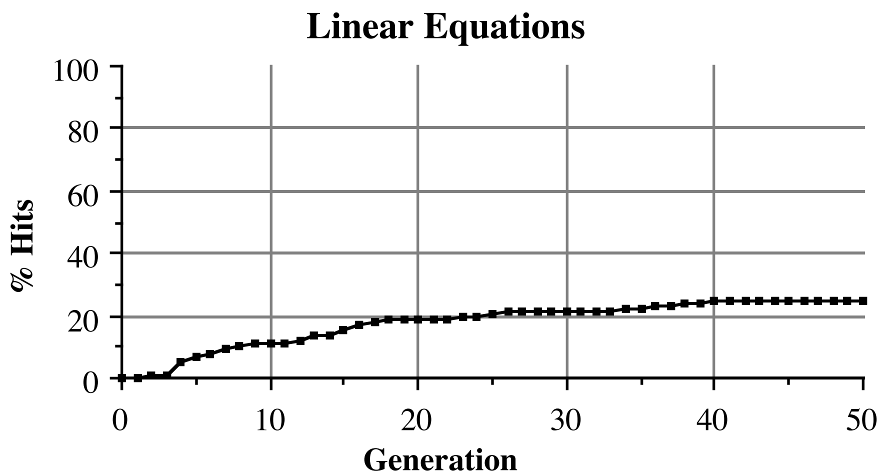
For the symbolic "data to function" differentiation problem (with $\text{Sin } x + x^2 + x$ as the function to be differentiated and $\text{Cos } x + 2x + 1$ as the desired result), the probability of success p_s that at least one individual S-expression comes within 1% of the derivative of target function for all 200 random points is displayed on the graph below as a function of the number of generations. In particular, the probability of success p_s after 50 generations is 36%.



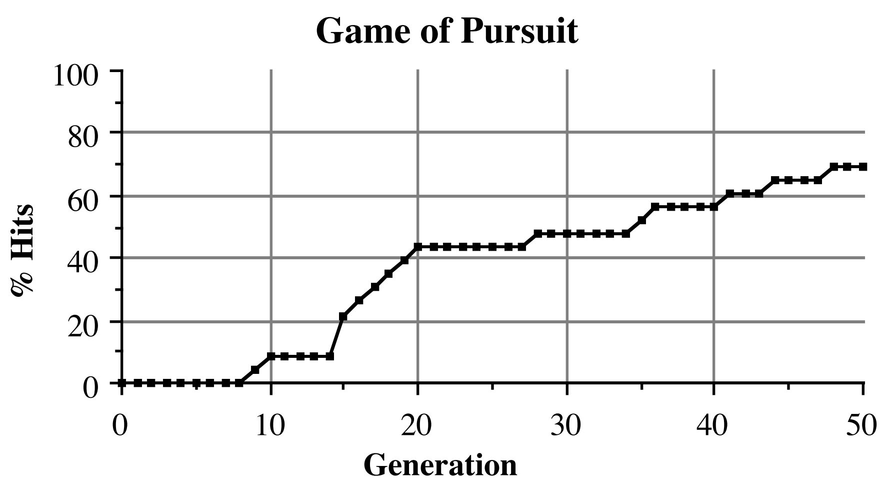
For the symbolic differential equation problem (dy/dx - 2y + 4x = 0), the probability of success p_s of finding at least one individual S-expression which, when substituted into the left hand side of the differential equation, comes within .01 of 0.0 (the right hand side of the equation) for all 200 random points is displayed on the graph below as a function of the number of generations. In particular, the probability of success p_s after 50 generations is 36%.



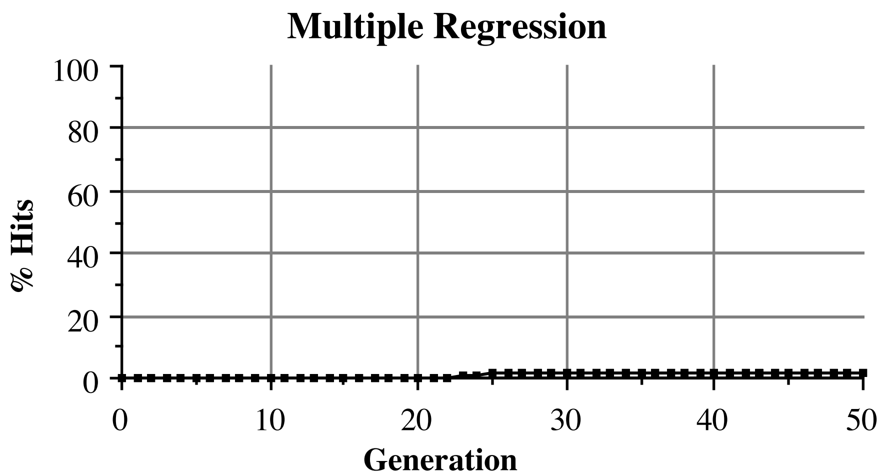
For the automatic programming problem (involving solving a pair of linear equations), the probability of success p_s that at least one individual S-expression which, comes within 1% of the solution for all 10 random equations is displayed on the graph below as a function of the number of generations. In particular, the probability of success p_s after 50 generations is 24%.



For the pursuer-evader differential game, the probability of success p_S that at least one individual pursuer strategy (S-expression) captures a minimax evader (within the "time out" limits) in all 10 environmental starting condition cases is displayed on the graph below as a function of the number of generations. In particular, the probability of success p_S after 50 generations is 69%.



For the symbolic multiple regression problem (with complex multiplication as the target function), the probability of success p_S that at least one individual pair of values (LIST of two S-expressions) comes within .01 of the target pair of values (as measured by the distance measure described herein) for all 50 random combinations of the four independent variables is displayed on the graph below as a function of the number of generations. In particular, the probability of success p_S after 50 generations is 2%.



Using the probability of success p_S by generation 51, we then computed the number of independent runs (niches) K needed to achieve the desired result with a probability of z = 99%. We then computed the number of individuals that must be processed in order to solve a particular problem by multiplying the number of runs K by 25,500 (i.e. population size 500 times number of generations 51). Note that the total number of individuals that need to be processed is a function of the percentage (z = 99%) and everything that affects the probability of success p_S .

The above method is far from perfect. Neither the fixed common population size nor the fixed common number of generations is optimal for any of the problems in the test suite. There are an infinity of alternative ways of distributing a fixed computational budget (i.e. total number of individuals to be processed) between the two primary parameters of the algorithm. For example, for some problems, a larger population size operating over proportionately fewer generations (with the same total number of individuals being processed) produces a higher percentage of success while for other problems, a tradeoff made in the opposite way produces a higher percentage of success.

In addition to the fact that neither the population size nor the number of generations is optimal, the number of individuals that must be processed in order to solve a particular problem is significantly overstated by the fact that we present the computer time as if each run was run the full 51 generations. In actual practice, we usually abort a run when it achieves the desired result. This overstatement is especially significant for those problems with a high percentage of success. The number of individuals that must be processed is also overstated because premature convergence is manifestly obvious in far fewer than 51 generations for most problems described herein and certain runs could be heuristically aborted to save computer time.

Nonetheless, this relatively simple method of estimation offers a rough estimate of the number of individuals that must be processed in order to solve a particular problem.

We also determined the amount of computer time required for each run of each problem on the Explorer II+™ computer. The vast majority of the computer time is consumed in fitness evaluation, rather than by the genetic operations or administrative tasks. Problems involving a simple fitness calculation over a relatively small number of environmental cases (e.g. the Fibonacci sequence, symbolic regression, etc.) take the smallest amount of time. Problems involving iterative DU loops (e.g. block stacking), iterative summation SIGMA loops, recursions, execution of a strategy over many time steps (e.g. artificial ant, differential pursuer-evader game) and optimal control problems (e.g. broom balancing) involving numerical integration over many time steps take considerably more time. These latter problems are especially time consuming in early generations when there are relatively more unfit individuals, including individuals which "time out" and therefore consume the maximum amount of time allowed by the problem.

The table below shows the probability of success p_s, the number of independent runs K (niches) likely to be needed to achieve the desired result with probability z = 99%, the likely number of individuals that must be processed (i.e. the number of niches times 500 times 51), the computer time per run (i.e. a full run of 51 generations), and the total computer time needed (i.e. the number of independent runs K times the computer time per run).

Problem	Percent of Runs Successful	Niches Needed	Total Individuals Processed	Time per Run (minutes)	Total Computer Time (minutes)
6-Multiplexer	68%	4	102,000	12	48
Block-Stacking	94%	2	51,000	666	1332
Artificial Ant	26%	15	382,500	46	690
Fibonacci Sequence	22%	19	484.500	12	228
Hofstadter Sequence	1%	458	11,679,000	8	3664
Symbolic Regression	31%	12	306,000	4	48
Symbolic Integration	20%	21	535,500	17	357
Symbolic Differentiation	36%	10	255,000	15	150
Differential Equations	1%	458	11,679,000	18	8244
Linear Equations	24%	16	408,000	10	160
Pursuer-Evader Game	69%	4	102,000	423	1692
Multiple Regression	2%	228	5,814,000	14	3192

Thus, we can describe the difficulty of a particular problem in terms of the likely number of niches (independent runs) required and the number of individual that must be processed. As can be seen from the table, the likely number of niches and likely number of individuals that must be processed to achieve the desired result with 99% probability ranges from 4 niches and 102,000 individuals (for the differential pursuer-evader game) to 151 niches and 3,850,500 individuals (for multiple regression of the complex multiplication operator).

A calculation of the expected number of processing steps would be a desirable addition to the reportage of the results of other algorithms in machine learning, neural networks, and artificial intelligence. For example, even non-probabalistic algorithms (such as the back propagation algorithm for neural network training) sometimes become trapped on local maxima or do not produce the desired result for other reasons. The number of such failed runs should affect the reported measures of the total number of processing steps required to execute the algorithm.

10. THEORETICAL DISCUSSION

Holland (1975) focused attention on the fact that genetic algorithms, in general, can be viewed as processing numerous schemata (similarity templates) rather than merely the relatively modest number of individuals in the population. For a string of length L over an alphabet of size k, a schema is identified by a string of length L over an extended alphabet consisting of the k alphabet symbols and the meta-symbol * (don't care) and consists of the set of individual strings from the population whose symbols match the symbols of the identifier for all specific positions (i.e. all positions except where the identifier has the "don't care" symbol). There are $(k+1)^L$ such schemata. Each individual string occurs in 2^L such schemata, regardless of k. Therefore, a population of only M individual strings makes up to $M*2^L$ appearances in the schemata (depending on population diversity). In the binary case, a schema of specificity O(H) corresponds to a hyperplane of dimensionality L-O(H) (containing $2^{L-O(H)}$ individuals) of the Hamming hypercube of dimension L. The average fitness f(H) of a schema H is the average of the observed fitness values of the individual strings contained in the schemata. This average fitness has an associated variance which depends on the number of items being summed to compute the average).

Holland showed that for genetic algorithms using fitness proportionate reproduction and crossover, the expected number of occurrences of every schema H in the next generation is approximately

$$m(H,t+1) >= \frac{f(H)}{f^*} m(H,t) (1 - \varepsilon)$$

where f* is the average fitness of the population and ε is small. In particular, viewed over several generations where either f(H)/f* is stationary or remains above 1 by at least a constant amount, this means that a schema with above-average (below-average) fitness appears in succeeding generations at an approximately exponentially increasing (decreasing) rate. Holland also showed that the mathematical form of the optimal allocation of trials among random variables in a multi-armed slot machine problem (involving minimizing losses while exploring new or seemingly non-optimal schemata, while also exploiting seemingly optimal schemata) is similarly approximately exponential so that the processing of schemata by genetic algorithms using fitness proportionate reproduction and crossover is mathematically near optimal. In particular, this allocation of trials is most nearly optimal when ε is small. For strings, ε is computed by dividing the defining length d(H) of the schema involved (i.e. the distance between the outermost specific, non-* symbols) by L-1 (i.e. the number of interstitial points where crossover may occur). Therefore, ε is short when d(H) is short (i.e. the schema is a small, short, compact "building block"). Thus, genetic algorithms process short-defining length schemata most favorably. More importantly, as a result, problems whose solutions can be incrementally "built up" from such small "building blocks" are most optimally handled by genetic algorithms.

In the genetic programming paradigm, the individuals in the population are LISP S-expressions (i.e. rooted point-labeled trees with ordered branches), instead of linear character strings. The set of similar individuals sharing common features (i.e. the schema) is a hyperspace of LISP S-expressions (i.e. a set of rooted point-labeled trees with ordered branches) sharing common features.

Consider first the case where the common features are a single sub-tree consisting of h specified points with no unspecified (don't care) points in that sub-tree. The set of individuals sharing the common feature is the hyperspace consisting of all rooted point-labeled trees in a plane containing the designated sub-tree as a sub-tree. This set of trees is infinite, but it can be partitioned into finite subsets by using the number of points in the tree as the partitioning parameter. If the subset of trees having a particular number of points and sharing a fully specified sub-tree is considered, fitness proportionate reproduction causes growth (or decay) in the size of that subset in the new population in accordance with the relative fitness of the subset to the average population fitness in the same near optimal way as it does for string-based linear genetic algorithms. Holland's results concerning the optimal allocation of trials and Holland's results concerning the growth (or decay) of the number of occurrences of schemata as a result of fitness proportionate reproduction (1975) do not depend on the character of the individual objects in the population. The deviation from this optimal rate of growth (or decay) of a schema is then caused by the crossover operation. The nature of this deviation for crossover does depend on the character of the individual objects in the population. For strings, this deviation is relatively small when the distances amongst the points defining the common feature is relatively small. The relevant measure of distance for strings is the maximum distance between any two defined positions.

The same principle appears to apply when the objects are trees. That is, the deviation is relatively small when the distances amongst the points defining the common feature is relatively small. In particular, when the specific positions of a schema are coextensive with a single sub-tree, the deviation is smallest. In particular, if ε is the ratio of the number of points in the sub-tree to the number of points in the tree, then ε is relatively small when the sub-tree is relatively small. Thus, for the case where the specific positions of a schema are coextensive with a single sub-tree, the overall effect of fitness proportionate reproduction and crossover is that subprograms (i.e. sub-trees, sub-lists) from relatively high fitness programs are used as "building blocks" for constructing new individuals in an approximately near optimal way. Over a period of time, this concentrates the search of the solution space into sub-hyperspaces of LISP S-expressions of ever decreasing dimensionality and ever increasing fitness.

This argument appears to extend to similarities defined by a sub-tree containing one or more non-specific points internal to the sub-tree and to similarities defined by a disjoint set of two or more sub-trees of either type. The deviation from optimality is relatively small to the extent that both the number of points defining the common feature is relatively small and the number of disjoint sub-trees is relatively small. Thus, the overall effect is that subprograms (i.e. sub-trees) from relatively high fitness individuals are used as "building blocks" for constructing new individuals.

The genetic programming paradigm is a natural extension of string-based linear genetic algorithms in another way. Genetic algorithms, in general, are mathematical algorithms which are based on Darwinian principles of reproduction and survival of the fittest and which transform a population of individuals (and their fitness in the environment) into a new population of individuals using operations analogous to genetic operations actually observed in nature. In this view, a character found at a particular position in a mathematical character string in a conventional string-based genetic algorithm is considered analogous to one of the four nucleotide bases (adenine, cytosine, guanine, or thymine) found in molecules of deoxyribonucleic acid (DNA). The observed fitness in the environment of the entire actual biological individual created using the passive information in a particular linear string of DNA is then used in the computation of average schema fitness for each schema represented by that individual. In contrast, the proactive computational procedure carried out by a LISP S-expression in the genetic programming paradigm can be viewed as analogous to the work performed by a protein in a living cell. The observed fitness in the environment of the entire actual biological individual created as a result of the action of the proactive LISP S-expressions contribute, in the same way as with string-based genetic algorithms, directly to the computation of average schema fitness for each schema represented by that individual. That is, the genetic programming paradigm employs the same automatic allocation of credit inherent in the basic string-based genetic algorithm described by Holland (1975) and inherent in Darwinian reproduction and survival of the fittest amongst biological populations in nature. This automatic allocation of credit contrasts with the connectionistic "bucket brigade" credit allocation and reinforcement algorithm used in classifier systems (Holland 1986, Holland and Reitman 1978) which is not founded on any observed natural mechanism involving adaptation amongst biological populations (Westerdale 1985).

11. FUTURE WORK

Since Holland's seminal 1975 work, many operational variations on the basic string-based genetic algorithm have been tested and studied. In a number of cases, a consensus has developed among the practitioners of genetic algorithms specifying a relatively standard approach to genetic algorithms as to those areas. In other areas, no such consensus exists. In any case, most of the operational issues affecting the basic string-based genetic algorithm reappear in connection with the genetic programming paradigm.

Thus, future work might include studying the tradeoffs, given a fixed computational budget, between the population size, the number of generations, and the number of niches (independent runs).

Future work might also include determining the optimum percentage of the population that should undergo crossover versus fitness proportionate reproduction on each generation; determining whether and how often the secondary genetic operations of mutation, permutation, editing, and "define building block" should be used; techniques for maintaining population diversity and minimizing premature convergence (cf. Booker 1987); the effect of the different ways to select the spouse of a given parent (i.e. proportionate to fitness versus at random); the use of probability distributions other than the uniform distribution for making the various random selections during the run; the use of non-standard fitness measures (e.g. the rank of the fitness value versus fitness in the standard probability form); the techniques for varying the parameters of the genetic algorithm during the run based on information gained from the run; the ways of parallelizing the algorithm; and the conditions for terminating the algorithm.

12. GENERAL DISCUSSION

The problem of automatically generating a computer program to solve a problem is generally viewed as an area that is unlikely to be fruitful.

This pessimistic view may stem, in part, from the personal experience of seeing how one small mistake in ones own computer programs can make the program completely inoperative. It is true that computer programs can be very sensitive to small errors. It is also true that an error as small as one nucleotide base among 2,870,000,000 nucleotide bases in the human genome can make the difference between normality and a fatal disease. However, the opposites to both of these observations are also true. For example, many small changes in the human genome (even in the portions of the deoxyribonucleic acid sequence that are actually translated into protein) merely produce slightly different, but functionally equivalent, proteins and have no net effect whatsoever. It is similarly true that many small changes in computer programs merely produce slightly different, but functionally equivalent, programs and have no net effect whatsoever. However, most people who have written computer programs vividly recall how particular small errors caused their programs not to work and so not similarly recall the many inconsequential changes that they have made in those same programs.

Another reason for the generally pessimistic view of the likelihood of solving the problem of automatic program generation is that one rarely focuses on the extent to which computer programs are built up from portions of code that the programmer has successfully used before. Indeed, most computer programs are replete with such "building blocks" previously used in other programs.

In summary, most people who have written computer programs overemphasize considerations that make computer programs seem unsusceptible to automatic generation and minimize the considerations on the other side of the argument.

Still another reason for the generally pessimistic view of the likelihood of solving the problem of automatic program generation is that standards are applied, as almost a knee jerk reaction, to the problem of program generation that are not applied to any other form of machine learning or artificial intelligence. One such standard can be stated as follows: "the paradigm must solve all problems of all types without any use of human judgment." Not surprisingly, when this standard is applied, the paradigm for program generation is found wanting. We want to make clear here that we do not claim that the genetic programming paradigm described in this paper meets this "straw man" standard. It does not. No other paradigms of machine learning, artificial intelligence, neural nets, genetic algorithms, or classifier systems claim to meet this "straw man standard." And, none do.

This "straw man" standard manifests itself is various ways. One manifestation is the "revelation" that the genetic programming paradigm described in this paper requires that a function set be specified by the user. This fact need not be "revealed." On the contrary, this paper makes it clear that the user of the genetic programming paradigm must select the function set for the problem. The genetic programming paradigm is similar to many other paradigms in machine learning in that a set of elementary functions must be specified. For example, ID3 and related algorithms for inducing decision trees start with a user-specified set of functions that test each training case (and actual case) for certain attributes. If the user of ID3 provides a set of such "attribute testing" functions for chess pieces that merely test whether they are made of wood or plastic, but do test their rank, ID3 will be not able to classify chess board positions as to whether they are winning or losing positions. Similarly, the user of SOAR provides the set of 24 elementary operations for moving the blank square in the 8-puzzle just as the users of other artificial intelligence paradigms for stacking blocks provide the elementary operations for moving individual blocks between the stack and the table.

Another manifestation of this straw man standard is the "revelation" that the user must select certain certain parameters, such as the population size and the number of generations. This fact need not be "revealed." On the contrary, this paper makes it clear that the user of the genetic programming paradigm must select certain parameters. The genetic programming paradigm is similar to many other paradigms in that certain parameters must be specified. For example, the user of a neural network must select the number of processing units for the input layer, the hidden layer(s), the output layer of the network, and, depending on the particular neural network paradigm that the user has selected, the thresholds, the biases, the nature of the connectivity allowed, and whether the network is feed-forward or recurrent. Similarly, the user of a classifier system must select the the number of bits in the condition parts of each if-then rule, the number of classifier rules allowed, the number of cycles to be run, and numerous other parameters. The choice of the number of bits in the condition parts of each if-then rule, in particular, limits the computational complexity of any solution that can be produced. In addition, the users of both neural networks and classifier systems must usually extensively preprocess the inputs from the external environment so that they are presented to the system in the manner that the paradigm can accept.

Another way that the "straw man" standard manifests itself concerns the assertion that if a paradigm is presented using an example for which the solution is already known, this knowledge somehow makes the solution less valid. All of the well-known paradigms in the literature have been presented using example problems to which the solution was already known. In fact, when example problems are presented for which the solution is not already known, the opposite question immediately arises, namely, the question as to how one knows that the paradigm has worked.

13. CONCLUSION

We have demonstrated how a number of seemingly different problems from artificial intelligence, symbolic processing, and machine learning can be reformulated as problems that require discovery of a computer program that produces a desired output for particular inputs. These problems include function learning, robotic planning, sequence induction, symbolic function identification, symbolic regression, symbolic "data to function" integration, symbolic "data to function" differentiation, solving differential equations, solving integral equations, finding inverse functions, solving general equations for numerical values, empirical discovery, concept formation, automatic programming, pattern recognition, optimal control, game-playing, multiple regression, and simultaneous architectural design and training of a neural network. We have then shown how such problems can be solved by genetically breeding computer programs using the genetic programming paradigm.

14. ACKNOWLEDGMENTS

Drs. Thomas Westerdale of Birkbeck College at the University of London, Martin A. Keane of Third Millennium Venture Capital Limited in Chicago, and John Perry (formerly of Texas Instruments Inc. in San Francisco) made numerous valuable comments on early drafts of this paper. Eric Mielke of the Texas Instruments Education Center in Austin implemented a method for programming the crossover operation in LISP that significantly improved execution time over the author's previous recursive version. James Rice of the Knowledge Systems Laboratory (KSL) at Stanford University in Palo Alto implemented a method for programming the crossover operation in LISP that significantly improved execution time over Mielke's, identified and corrected several illusive programming problems, and greatly expanded the efficiency and functionality of the author’s program (particularly its interfaces and instrumentation) in numerous extremely helpful respects. John Holland of the University of Michigan, Stewart Wilson of the Rowland Institute of Cambridge, Massachusetts, Lawrence Davis of BBN Inc. of Cambridge, and James Rice made numerous valuable comments on recent drafts of this paper.

15. REFERENCES

Anderson, Charles W. Learning to control and inverted pendulum using neural networks. IEEE Control Systems Magazine. 9(3). Pages 31-37. April 1989.

Axelrod, R. The evolution of strategies in the iterated prisoner's dilemma. In Davis, Lawrence (editor) Genetic Algorithms and Simulated Annealing London: Pittman 1987.

Barto, A. G., Anandan, P., and Anderson, C. W. Cooperativity in networks of pattern recognizing stochastic learning automata.In Narendra,K.S. Adaptive and Learning Systems. New York: Plenum 1985.

Booker, Lashon Improving search in genetic algorithms. In Davis, Lawrence (editor) Genetic Algorithms and Simulated Annealing London: Pittman 1987.

Booker, Lashon, Goldberg, David E., and Holland, John H. Classifier systems and genetic algorithms. Artificial Intelligence 40 (1989) 235-282.

Citibank, N. A. CITIBASE: Citibank Economic Database (Machine Readable Magnetic Data File), 1946-Present. New York: Citibank N.A. 1989.

Cramer, Nichael Lynn. A representation for the adaptive generation of simple sequential programs. Proceedings of an International Conference on Genetic Algorithms and Their Applications. Hillsdale, NJ: Lawrence Erlbaum Associates 1985.

Davis, Lawrence (editor) Genetic Algorithms and Simulated Annealing London: Pittman 1987.

Davis, Lawrence and Steenstrup, M. Genetic algorithms and simulated annealing: An overview. In Davis, Lawrence (editor) Genetic Algorithms and Simulated Annealing London: Pittman 1987.

Dawkins, Richard. The Blind Watchmaker. New York: W. W. Norton 1987.

De Jong, Kenneth A. Genetic algorithms: A 10 year perspective. Proceedings of an International Conference on Genetic Algorithms and Their Applications. Hillsdale, NJ: Lawrence Erlbaum Associates 1985.

De Jong, Kenneth A. On using genetic algorithms to search program spaces. Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms. Hillsdale, NJ: Lawrence Erlbaum Associates 1987.

De Jong, Kenneth A. Learning with genetic algorithms: an overview. Machine Learning, 3(2), 121-138, 1988.

Doan, Thomas A. User Manual for RATS - Regression Analysis of Time Series. Evanston, IL: VAR Econometrics, Inc. 1989

Fogel, L. J., Owens, A. J. and Walsh, M. J. Artificial Intelligence through Simulated Evolution. New York: John Wiley 1966.

Friedberg, R. M. A learning machine: Part I. IBM Journal of Research and Development, 2(1), 2-13, 1958.

Friedberg, R. M. Dunham, B. and North, J. H. A learning machine: Part II. IBM Journal of Research and Development, 3(3), 282-287, 1959.

Fujiki, Cory and Dickinson, John. Using the genetic algorithm to generate LISP source code to solve the prisoner's dilemma. In Grefenstette, John J.(editor). Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms. Hillsdale, NJ: Lawrence Erlbaum Associates 1987.

Fujiki, Cory. An Evaluation of Holland's Genetic Algorithm Applied to a Program Generator. Master of Science Thesis, Department of Computer Science, Moscow, ID: University of Idaho, l986.

Goldberg, David E. Computer-Aided Gas Pipeline Operation Using Genetic Algorithms and Rule Learning. PhD dissertation. Ann Arbor: University of Michigan. 1983.

Goldberg, David E. Genetic Algorithms in Search, Optimization, and Machine Learning. Reading, MA: Addison-Wesley l989.

Goldberg, David E. and Holland, John H. (editors). Special issue on genetic algorithms. Machine Learning, volume 3, numbers 2/3, 1988.

Goldberg, David E. Sizing populations for serial and parallel genetic algorithms. In Schaffer, J. D. (editor) Proceedings of the Third International Conference on Genetic Algorithms. San Mateo, CA: Morgan Kaufmann Publishers Inc. 1989.

Green, Cordell C. et. al. Progress Report on Program-Understanding Systems. Stanford Artificial Intelligence Laboratory memo AIM-240. Stanford University Computer Science Department. August 1974.

Grefenstette, John J.(editor). Proceedings of an International Conference on Genetic Algorithms and Their Applications. Hillsdale, NJ: Lawrence Erlbaum Associates l985.

Grefenstette, John J.(editor). Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms. Hillsdale, NJ: Lawrence Erlbaum Associates l987a.

Grefenstette, John J. Incorporating problem specific knowledge into Genetic Algorithms. In Davis, L. (editor) Genetic Algorithms and Simulated Annealing London: Pittman l987b.

Hallman, Jeffrey J., Porter, Richard D., Small, David H. M2 per Unit of Potential GNP as an Anchor for the Price Level. Washington,DC: Board of Governors of the Federal Reserve System. Staff Study 157, April 1989.

Hicklin, Joseph F., Application of the Genetic Algorithm to Automatic Program Generation. Master of Science Thesis, Department of Computer Science. Moscow, ID: University of Idaho l986.

Hillis, W. Daniel. Co-evolving parasites improve simulated evolution as an optimization procedure. In Forrest, Stephanie (editor) Emergent Computation: Self-organizing, Collective, and Cooperative Computing Networks. Cambridge, MA: MIT Press 1990 (to appear). Also in Physica D 1990 (to appear).

Hillis, W. Daniel. "Co-evolving parasites improve simulated evolution as an optimization procedure" In Langton, Christopher G. and Farmer, J. Doyne. (editors) Proceedings of the Second Conference on Artificial Life. Redwood City, CA; Addison-Wesley 1990. In press.

Hinton, Geoffrey. Neural Networks for Artificial Intelligence. Santa Monica, CA: Technology Transfer Institute. Documentation dated December 12, 1988.

Hinton, Geoffrey. Connectionist learning procedures. Artificial Intelligence. 40 (1989) 185-234.

Holland, John H. Adaptation in Natural and Artificial Systems, Ann Arbor, MI: University of Michigan Press 1975.

Holland, John H. Escaping brittleness: The possibilities of general-purpose learning algorithms applied to parallel rule-based systems. In Michalski, Ryszard S., Carbonell, Jaime G. and Mitchell, Tom M. Machine Learning: An Artificial Intelligence Approach, Volume II. P. 593-623. Los Altos, CA: Morgan Kaufman l986.

Holland, John H. ECHO: Explorations of Evolution in a Minature World. In Langton, Christopher G. and Farmer, J. Doyne. (editors) Proceedings of the Proceedings of the Second Conference on Artificial Life. Redwood City, CA; Addison-Wesley 1990. In press.

Holland, John H. and Burks, Arthur W. Adaptive Computing System Capable of Learning and Discovery. U. S. Patent 4,697,242. Issued September 29, 1987.

Holland, John H. and Burks, Arthur W. Method of controlling a Classifier System. U.S. Patent 4,881,178. Issued November 14, 1989.

Holland, John H, Holyoak, K.J., Nisbett, R.E., and Thagard, P.A. Induction: Processes of Inference, Learning, and Discovery. Cambridge, MA: MIT Press l986.

Holland, John H. and Reitman, J.S. "Cognitive Systems Based on Adaptive Algorithms". In Waterman, D. A. and Hayes-Roth, Frederick, Pattern-Directed Inference Systems, New York: Academic Press l978.

Humphrey, Thomas M. Precursors of the P-star model. Economic Review. Richmond, VA: Federal Reserve Bank of Richmond. July-August 1989. Pages 3-9.

Isaacs, Rufus. Differential Games. New York: John Wiley 1965.

Jefferson, David, Collins, Rob, et. al. The Genesys System: Evolution as a theme in artificial life. In Langton, Christopher G. and Farmer, D. (editors) Proceedings of Second Conference on Artificial Life. Redwood City, CA: Addison-Wesley. 1990. In Press.

Koza, John R. Hierarchical genetic algorithms operating on populations of computer programs. Proceedings of the 11th International Joint Conference on Artificial Intelligence (IJCAI). San Mateo, CA: Morgan Kaufman 1989.

Koza, John R. "A genetic approach to econometric modeling." Sixth World Congress of the Econometric Society. Barcelona, Spain. August 1990. In Press. 1990a.

Koza, John R. Econometric modeling by genetic breeding of mathematical functions. Proceedings of International Symposium on Economic Modeling. Urbino, Italy: 1990. In Press. 1990b.

Koza, John R. Non-Linear Genetic Algorithms for Solving Problems. U. S. Patent 4,935,877. Filed May 20, 1988. Issued June 19, 1990. 1990c.

Koza, John R. Non-Linear Genetic Algorithms for Solving Problems by Finding A Fit Composition of Functions. U. S. Patent Continuation-In-Part Application. Filed March 28, 1990. 1990d.

Koza, John R. and Keane, Martin A. Cart centering and broom balancing by genetically breeding populations of control strategy programs. Proceedings of International Joint Conference on Neural Networks, Washington, January 1990. Volume I. 1990a.

Koza, John R. and Keane, Martin A. Genetic breeding of non-linear optimal control strategies for broom balancing. Proceedings of the Ninth International Conference on Analysis and Optimization of Systems, Antibes, France, June 1990. New York: Spring-Verlag 1990b.

Laird, John, Rosenbloom, Paul, and Newell, Allen. SOAR: Universal Subgoaling and Chunking. Boston, Ma: Kluwer Academic Publishers 1986.

Langley, Pat and Zytkow, Jan M. Data-driven approaches to empirical discovery. Artificial Intelligence. 40 (1989) 283-312.

Langley, Pat, Simon, Herbert A., Bradshaw, Gary L., and Zytkow, Jan M. Scientific Discovery: Computational Explorations of the Creative Process. Cambridge, MA: MIT Press, 1987.

Lenat, Douglas B. AM: An Artificial Intelligence Approach to Discovery in Mathematics as Heuristic Search. PhD Dissertation. Computer Science Department. Stanford University. 1976.

Lenat, Douglas B. The role of heuristics in learning by discovery: Three case studies. In Michalski, Ryszard S., Carbonell, Jaime G. and Mitchell, Tom M. Machine Learning: An Artificial Intelligence Approach, Volume I. P. 243-306. Los Altos, CA: Morgan Kaufman l983.

Lenat, Douglas B. and Brown, John Seely. Why AM and EURISKO appear to work. Artificial Intelligence. 23 (1984). 269-294.

Magurran, Anne E. Ecological Diversity and Its Measurement. Princeton, NJ: Princeton University Press. 1988.

Miller, John H. The Co-evolution of Automata in the Repeated Prisoner's Dilemma. Sante Fe Institute Report 89-003. 1989.

Miller, John H. The evolution of automata in the repeated prisoner's dilemma. In Two Essays on the Economics of Imperfect Information. PhD dissertation, Department of Economics, University of Michigan, 1988.

Mills, R. D. Using a small algebraic manipulation system to solve differential and integral equations by variational and approximation techniques. Journal of Symbolic Computation. 3(3). 291-301. 1987.

Minsky, Marvin L. and Papert, Seymour A. Perceptrons. Cambridge, MA: The MIT Press. 1969.

Nilsson, Nils J. Principles of Artificial Intelligence. Los Altos, CA: Morgan Kaufman l980.

Nilsson, Nils J. Action networks. Draft Stanford Computer Science Department Working Paper, October 24, 1988. Stanford, CA: Stanford University. 1988a.

Nilsson, Nils J. Private communication. 1988b.

Quinlan, J. R. Induction of decision trees. Machine Learning 1 (1), 81-106, 1986.

Quinlan, J. R. An empirical comparison of genetic and decision-tree classifiers. Proceedings of the Fifth International Conference on Machine Learning. San Mateo, CA: Morgan Kaufmann. 1988.

Rice, James P. Private communication. 1989.

Robertson, George. Parallel implementation of genetic algorithms in a classifier system. In Davis, L. (editor) Genetic Algorithms and Simulated Annealing London: Pittman l987.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. Learning internal representations by error propagation. In Rumelhart, D. E., McClelland, J. L., et. al. Parallel Distributed Processing. Volume 1, Chapter 8. Cambridge, MA: The MIT Press. 1986.

Samuel, A. L. Some studies in machine learning using the game of checkers. IBM Journal of Research and Development, 3(3), 210-229 (1959).

Schaffer, J. D. Some effects of selection procedures on hyperplane sampling by genetic algorithms. In Davis, L. (editor) Genetic Algorithms and Simulated Annealing London: Pittman l987.

Schaffer, J. D. (editor) Proceedings of the Third International Conference on Genetic Algorithms. San Mateo, CA: Morgan Kaufmann Publishers Inc. 1989.

Smith, Steven F. A Learning System Based on Genetic Adaptive Algorithms. PhD dissertation. Pittsburgh: University of Pittsburgh 1980.

Smith, Steven F. Flexible learning of problem solving heuristics through adaptive search. Proceeding of the Eighth International Conference on Artificial Intelligence. Karlsruhe, Germany: Morgan Kaufman l983.

Steele, Guy L. Jr. Common LISP. Digital Press. 1984.

Tanese, Reiko. Distributed Genetic Algorithm for Function Optimization. PhD. dissertation. Department of Electrical Engineering and Computer Science. University of Michigan. 1989.

Westerdale, Thomas H. The bucket brigade is not genetic. Proceedings of an Internal Conference on Genetic Algorithms and Their Applications. Hillsdale, NJ: Lawrence Erlbaum Associates l985.

Widrow, Bernard. Pattern recognizing control systems. Computer and Information Sciences

(COINS) Symposium Proceedings. Washington, DC: Spartan Books, 1963.

Widrow, Bernard. The original adaptive neural net broom balancer. 1987 IEEE International Symposium on Circuits and Systems. Vol. 2.

Wilson, Stewart. W. Classifier Systems and the animat problem. Machine Learning, 3(2), 199-228, 1987a.

Wilson, Stewart. W. Hierarchical credit allocation in a classifier system. Proceedings of the Tenth International Joint Conference on Artificial Intelligence, 217-220, 1987b.

Wilson, Stewart W. Bid competition and specificity reconsidered. Journal of Complex Systems. 2(6), 705-723, 1988.

Wolfram, Stephen. Mathematica™ - a system for doing mathematics by computer. Redwood City, CA: Addison-Wesley 1988.