ISSN:0975-9646

Shresth Joshi et al, / (IJCSIT) International Journal of Computer Science and Information Technologies, Vol. 3 (6), 2012,5359-5368

Genetically Breeding Populations of Computer Programs to Solve Problems

Shresth Joshi, Priya Tomar, Dharmendra Kelde
Department of Information Technology
Shri Dadaji Institute of Technology and Science, Khandwa

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 pursuerevader games; neural network design; and finding a game-playing strategy for a discrete game in extensive form.

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.INTRODUCTION:

Genetic programming is a technique pioneered by John Koza which enables computers to solve problems without being explicitly programmed. It works by using John Holland's genetic algorithms to automatically generate computer programs. Genetic algorithms were devised by Holland as a way of harnessing the power of natural evolution for use within computers. Natural evolution has seen the development of complex organisms (e.g. plants and animals) from simpler single celled life forms. Holland's GAs are simple models of the essentials of natural evolution and inheritance.

The growth of plants and animals from seeds or eggs is primarily controlled by the genes they inherited from their parents. The genes are stored on one or more strands of DNA. In asexual reproduction the DNA is a copy of the parent's DNA, possibly with some random changes, known as mutations. In sexual reproduction, DNA from both parents is inherited by the new individual. Often about half of each parent's DNA is copied to the child where it joins with DNA copied from the other parent. The child's DNA is usually di_erent from that in either parent.

Natural Evolution arises as only the fittest individuals survive to reproduce and so pass on their DNA to subsequent generations. That is DNA which produces fitter individuals is likely to increase in proportion in the population. As the DNA within the population changes, the species as a whole changes, i.e. it evolves as a result of selective survival of the individuals of which it is composed.

Genetic algorithms contain a population of trial solutions to a problem, typically each individual in the population is modeled by a string representing its DNA. This population is "evolved" by repeatedly selecting the "fitter" solutions and producing new solution from them. The new solutions replacing existing solutions in the population. New individuals are created either asexually (i.e. copying the string) or sexually (i.e. creating a new string from parts of two parent strings).In genetic programming the individuals in the population are computer programs. To ease the process of creating new programs from two parent programs, the programs are written as trees. New programs are produced by removing branches from one tree and inserting them into another. This simple process ensures that the new program is also a tree and so is also syntactically valid.

As an example, suppose we wish a genetic program to calculate y = x2. Our population of programs might contain a program which calculates y = 2x x (see _gure 1) and another which calculates y = xx x x3 x (_gure 2). Both are selected from the population because they produce answers similar to y = x2 (_gure 4), i.e. they are of high _tness. When a selected branch (shown shaded) is moved from the father program and inserted in the mother (displacing the existing branch, also shown shaded) a new program is produced which may have even high _tness. In this case the resulting program (_gure 3) actually calculates y = x2 and so this program is the output of our GP. The remainder of this paper describes genetic algorithms in more detail, placing them in the context of search techniques, then explains genetic programming, its history, the _ve steps to GP, shows these steps being used in our example and gives a taxonomy of current GP research and applications. Current GP research and applications are presented in some detail.

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 evolutionaryprocess. In this regard, it contrasts sharply with numerous other efforts to solve adaptive systems problem by merely Òsaving and mutating the bestÓ, such as the 1966 *Artificial Intelligence through Simulated Evolution* (Fogel et. al.) and other work using only asexual mutation .

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.

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 = {f1, f2, ... , fn} and the available set of m terminals T = {a1, a2, ... , am}. Each particular function f in F takes a specified number z(f) of arguments b1, b2, ..., bz(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 1950Õs) 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.

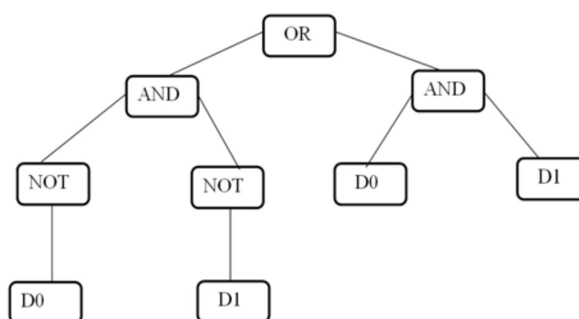
Fifth, the LISP programming language is reentrant.

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 Sexpression, the addition function (+) appears just inside the left-most parenthesis of the Sexpression. 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 ÒfunctionÓ 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 È 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 10 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 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 nonuniform distribution or by seeding particular individuals into the population.

3.3. THE OPERATIONS THAT MODIFY THE STRUCTURES

The two primary operations for modifying the structures undergoing adaptation are Darwinian fitness proportionate reproduction and crossover (recombination).

3.3.1 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(si(t)) is the fitness of individual si in the population at generation t, then, each time this operation is performed, each individual in the population has a probability of being copied into the next generation by the operation of fitness proportionate reproduction.

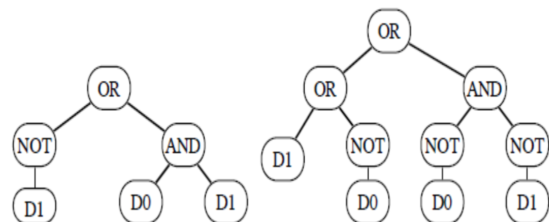
$$\frac{f(s_i(t))}{\sum_{j=1}^{M} f(s_j(t))}$$

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.

3.3.2 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.



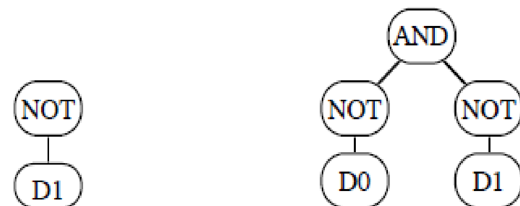
In terms of LISP S-expressions, the two parents are

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

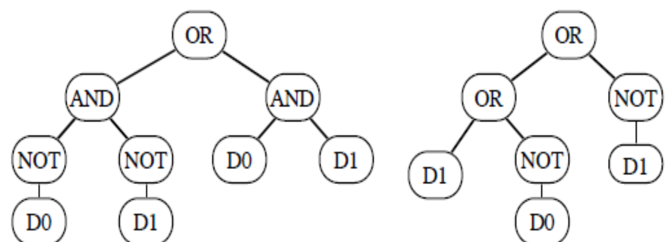
and

(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) isselected 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.

3.4 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.

3.5 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.

3.6 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 10616 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.

4. 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.

4.1. 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 p in a particular problem where there is a possibility that this particular constant would be useful. Of course, if one failed to include p 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.

4.2. 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 Ò:infinityÓ. 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 (ÒprogramÓ) 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 ex.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.

4.3. 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.

4.4. 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 Sexpression 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).

4.5. 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 .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.

4.6. 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.

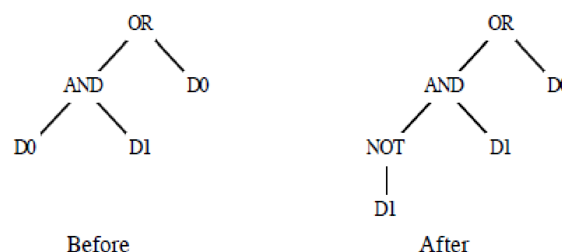
The extent to which one uses such domain specific heuristics is, of course, dependent on whether the primary objective is to solve a specific problem at hand or to study the process in the purest theoretical form. We have chosen not to use such domain specific heuristics in the work reported here.

5. 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

5.1. 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 subtree 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.



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).

5.2. 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.

5.3. 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 Sexpression. 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.

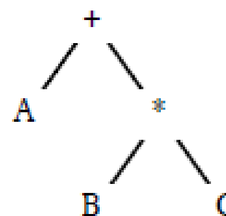
The main reason for the editing operation is convenience. It simplifies S-expressions and saves computer resources. It also appears to improve overall performance slightly. The editing operation apparently improves performance by reducing the vulnerability of an S-expression to disruption due to crossover 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.

5.4. THE ÒDEFINE BUILDING BLOCKÓ OPERATION

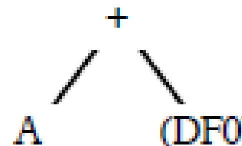
The “define building block” operation is a means for automatically identifying potentially useful “building blocks” while the algorithm is running. The “define building block” operation is an asexual operation in that it operates on only one parental S-expression. The individual is selected proportional 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, Ò(DF0)Ó 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 Òbuilding blockÓ 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-ex-pression may participate in additional genetic operations during the current generation, including replication (fitness proportionate reproduction), crossover (recombination), or even another "define building block" operation.

6. 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 x2 +x +2; however, at generation 10, the environment abruptly changes to the cubic polynomial x3 + x2 +2x +1; and, at generation 20, it changes again to a new quadratic polynomial x2 +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.

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.

REFERENCES:

1. Anderson, Charles W. Learning to control and inverted pendulum using neural networks. IEEE Control Systems Magazine. 9(3). Pages 3l-37. April l989.
2. Axelrod, R. The evolution of strategies in the iterated prisonerÕs dilemma. In Davis, Lawrence (editor) Genetic Algorithms and Simulated Annealing London: Pittman l987.
3. 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.
4. Booker, Lashon Improving search in genetic algorithms. In Davis, Lawrence (editor) Genetic Algorithms and Simulated Annealing London: Pittman l987.
5. Booker, Lashon, Goldberg, David E., and Holland, John H. Classifier systems and genetic algorithms. Artificial Intelligence 40 (1989) 235-282.
6. Citibank, N. A. CITIBASE: Citibank Economic Database (Machine Readable Magnetic Data File), 1946-Present. New York: Citibank N.A. 1989.
7. 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 l985.
8. Davis, Lawrence (editor) Genetic Algorithms and Simulated Annealing London: Pittman l987.
9. Davis, Lawrence and Steenstrup, M. Genetic algorithms and simulated annealing: An overview. In
10. Davis, Lawrence (editor) Genetic Algorithms and Simulated Annealing London: Pittman l987.
11. Dawkins, Richard. The Blind Watchmaker. New York: W. W. Norton 1987.
12. De Jong, Kenneth A. Genetic algorithms: A l0 year perspective. Proceedings of an International Conference on Genetic Algorithms and Their Applications. Hillsdale, NJ: Lawrence ErlbaumAssociates l985.
13. 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 l987.
14. 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.
15. Friedberg, R. M. A learning machine: Part I. IBM Journal of Research and Development, 2(1), 2-13, l958. Friedberg, R. M. Dunham, B. and North, J. H. A learning machine: Part II. IBM Journal of Research and Development, 3(3), 282-287, l959.
16. 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 l987.
17. Fujiki, Cory. An Evaluation of Holland's Genetic Algorithm Applied to a Program Generator.
18. Master of Science Thesis, Department of Computer Science, Moscow, ID: University of Idaho.