This is Microsoft Word 5 version of original MacWrite 2 text submitted to typesetter to make final camera-ready version of IJCAI-89 paper.

Operating on Programs

John R. Koza
Computer Science Department
Stanford University
Stanford, California 94305

Abstract

Existing approaches to artificial intelligence problems such as sequence induction, automatic programming, machine learning, planning, and pattern recognition typically require specification in advance of the size and shape of the solution to the problem (often in a unnatural and difficult way). This paper reports on a new approach in which the size and shape of the solution to such problems is dynamically created using Darwinian principles of reproduction and survival of the fittest. Moreover, the resulting solution is inherently hierarchical. The paper describes computer

experiments, using the author's 4341 line LISP program, in five areas of artifical intelligence, namely (1) sequence induction (e.g. inducing a computational procedure for the recursive Fibonacci sequence and inducing a computational procedure for a cubic polynomial sequence), (2) automatic programming (e.g. discovering a computational procedure for solving pairs of linear equations, solving quadratic equations for complex roots, and discovering trigonometric identities), (3) machine learning of functions (e.g. learning a Boolean multiplexer function previously studied in neural net and classifier system work and learning the exclusive-or and parity function), (4) planning (e.g. developing a robotic action sequence that can stack an arbitrary initial configuration of blocks into a specified order), and (5) pattern recognition (e.g. translation-invariant recognition of a simple one dimensional shape in a linear retina).

1 Introduction

Sequence induction requires developing a computational procedure that can generate any arbitrary element in a sequence $S = S_0, S_1, ..., S_j, ...$ given a finite number of specific examples of the values of the sequence. Examples are finding a correct recursive computational procedure for the Fibonacci sequence or finding a polynomial sequence expression of the appropriate order given a finite sampling of the

initial values of the sequence. Although induction problems admittedly do not have closed mathematical solutions, the ability to correctly perform induction is widely accepted as a component of human intelligence.

Automatic programming requires developing a computer program that can produce a desired output for a given set of inputs. Examples include finding 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 numbers x_1 and x_2, finding a computational procedure for solving a given quadratic equation $ax^2 + bx + c = 0$ for complex-valued roots x_1 and x_2, and solving trigonometric identities.

Machine learning of a function requires developing a computational procedure that can return the correct functional value for any combination of arguments given a finite number of specific examples of particular combinations of arguments and the associated functional value. An example is the problem of learning the Boolean multiplexer function. The Boolean multiplexer function has been repeatedly used as a test function in studies of neural nets (Barto et. al. 1985) and classifier systems (Wilson 1987a). Another example is the parity function.

Planning in artificial intelligence and robotics requires finding a plan that

receives information from sensors about the state of various objects in robotic environment and uses that information to select a sequence of functions to execute in order to change the state of the objects in the robotic environment. An example of a planning problem involves generating a general plan for stacking labeled blocks onto a target tower in a specified desired order, given an arbitrary initial configuration of blocks.

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.

All of these problems, and many similar problems in artifical intelligence and symbolic processing, can be viewed as requiring the creation of a LISP S-expression (i.e. a computer program, a computational procedure, a robotic plan) comprised of various functions and various atoms appropriate to the given problem domain that returns the desired values (and performs the desired side effects) when presented with a particular combination of input values.

In each case, it would be difficult and unnatural to try to specify the size and shape of the eventual solution in advance. Moreover, attempting such specification in advance narrows the window by which the system views the world and may well preclude finding the solution.

The fitness of any LISP S-expression in a problem environment can be naturally measured by the sum of the distances (taken for all the cases in the test suite) between the point in the solution space (whether Boolean-valued, integer-valued, real-valued, vector-valued, or complex-valued) created by the S-expression for a given set of arguments and the correct point in the solution space. The closer this sum is to zero, the better the S-expression.

As will be seen, the LISP S-expression required to solve the problem will, in each case, emerge from a simulated evolutionary process which starts with an initial population of randomly generated LISP S-expressions containing functions and atoms appropriate to the problem domain. Predictably, these initial random individual S-expressions will have exceedingly low fitness (when measured by the previously mentioned objective function). Nonetheless, some individuals in the population will be somewhat more fit in the environment than others. Then, a process of sexual reproduction among two parental S-expression selected in proportion to fitness creates offspring S-expressions comprised of sub-expressions ("building blocks") from their parents. The offspring then replace their parents. At each stage, the only input is the fitness of the individuals in the current population. This process tends to produce populations which, over a period of

generations, exhibit increasing average fitness in dealing with their environment and which also can robustly adapt to changes in their environment.

2 Background

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 character strings in Adaptation in Natural and Artificial Systems (Holland 1975).

Although genetic algorithms superficially seem to only process the particular individual binary strings present in the current population, Holland's 1975 work focused attention on the fact that they actually also automatically process large amounts of useful information in parallel concerning unseen Boolean hyperplanes (called similarity templates or schemata) representing numerous similar individuals not actually present in the current population. Genetic algorithms have a property of "intrinsic parallelism" which enable them to create individual strings for the new population in such a way that all the hyperplanes representing similar other individuals are all automatically

expected to be represented (without any explicit computation or memory beyond the population itself) in proportion to the fitness of the hyperplane relative to the average population fitness. 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 when the process is viewed as a set of multi-armed slot machine problems requiring an optimal allocation of trials.

Holland's 1975 work also highlighted the relative unimportance of mutation in the evolutionary process and contrasts sharply in this regard with numerous other efforts based on the approach of merely saving the best from among asexual random mutants, such as the 1966 Artificial Intelligence through Simulated Evolution (Fogel et. al. 1966) and other work (Lenat 1983, Hicklin 1986).

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 (De Jong 1987, Smith 1980, Fujuki 1986, Hicklin 1986, Cramer 1985). 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). 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 (e.g. the Fibonacci sequence). Moreover, string-based representation schemes do not facilitate computer programs modifying themselves and then executing themselves. Moreover, without dynamic variability, 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.

3 Hierarhical Genetic Algorithms

The LISP programming language is especially well-suited for handling hierarchies, recursions, logical functions, compositions of functions, self-modifying computer programs, self-

executing computer programs, iterations, late typing of variables and expressions, and complex structures whose size and shape is dynamically determined (rather than predetermined in advance). Because of these features, the LISP programming language allows the creation of "hierarchical" genetic algorithms for simulated evolution in which the population consists of individual hierarchical LISP S-expressions, rather than strings of characters or other objects (whether of fixed or variable length).

In hierarchical genetic algorithms, the set of possible S-expressions for a particular domain of interest depends on the functions and atoms that are available in the domain. The possible S-expressions are those that can be composed recursively from a set of n functions $F = \{f_1, f_2, ..., f_n\}$ and a set of m atoms $A = \{a_1, a_2, ..., a_m\}$. Each particular function f in F takes a specified number z(f) of arguments $b_1, b_2, ..., b_{z(f)}$. For example, the LISP S-expression (+ (σ (- J 1) 1) (σ (- J (+ 1 1) 0))) is an S-expression for the Fibonacci sequence. In this representation, J is the index for the current sequence element and σ(x,y) is the sequence referencing function returning the value of the sequence at position x (provided x is between 0 and J-1) or the default value y (if σ is being asked to provide a position of the sequence that is not yet defined).

The operation of fitness proportionate reproduction for hierarchical genetic algorithms is the basic engine of Darwinian reproduction and survival of the fittest. It is an asexual operation in that it operates on only one parental S-expression. The result of this operation is one offspring S-expression. In this operation, if $s_i(t)$ is an individual in the population at generation t with fitness value $f(s_i(t))$, it will be copied into the mating pool for the next generation with probability $f(s_i(t))/\Sigma f(s_i(t))$.

The crossover operation is a sexual operation that starts with two parental S-expressions. Its result is, for convenience, two offspring S-expressions. Every LISP S-expression can be depicted graphically as a rooted point-labeled tree in a plane whose internal points are labeled with functions, whose external points (leaves) are labeled with atoms, and whose root is labeled with the function (or atom) appearing just inside the outermost left parenthesis. The crossover operation begins by randomly and independently selecting one point in each parent using a uniform probability distribution. This crossover operation is well-defined for any two S-expressions and any two crossover points and the resulting offspring are always valid LISP S-expressions. Offspring contain some traits 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 this parent). 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.

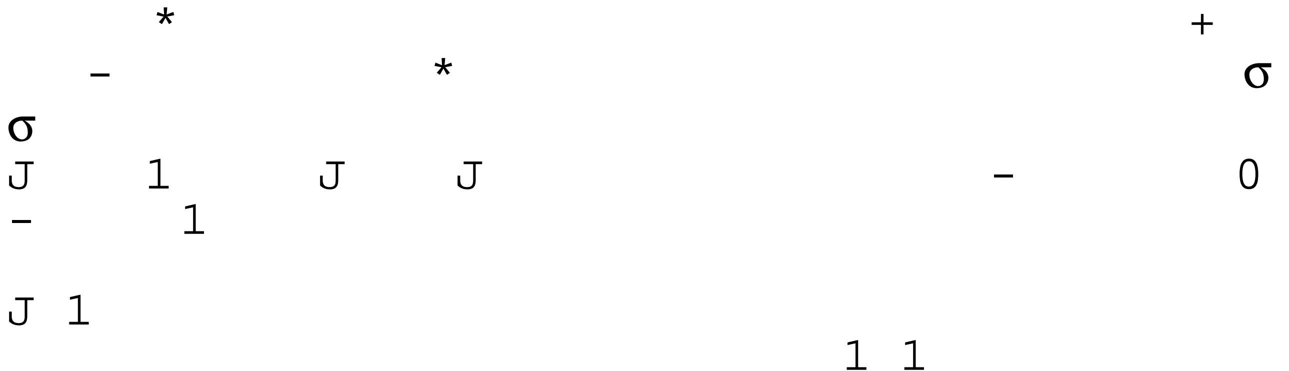
For example, consider the two parental LISP S-expressions below.

```
              *
+
     σ                *                        σ
-
  -      1            J   J                   -
0      J   1
J   1                                  J   +
                                          1 1
```

Assume that the points of trees are numbered in a depth-first way starting at the left. Suppose that point 2 (out of the 9 points of the first parent) was selected as the crossover point for the first parent (i.e. the σ) and that point 9 (out of the 11 points of the second parent) was selected as the crossover point of the second parent (i.e. the subtraction function - at the right). The two crossover fragments are below.

In terms of LISP S-expressions, the two parents are (* (σ (- J 1) 1) (* J J))) and (+ (σ (- J (+ 1 1)) 0) (- J 1)) and the two crossover fragments are the underlined sublists.

The two offspring resulting from crossover are shown below.



Note that the second offspring above is a perfect solution for the Fibonacci

sequence, namely (+ (σ (- J (+ 1 1) 0)) (σ (- J 1) 1)).

Crossover can be efficiently implemented in LISP using the RPLACA function in LISP (in conjunction with the COPY-TREE function) so as to destructively change the pointer of the CONS cell at the crossover point of one parent so that it points to the crossover fragment (sublist) of the other parent.

In each of the runs reported herein, between 75% and 80% of the crossover points are restricted to function (internal) points of the tree in order to promote the recombining of larger structures than is the case with an unrestricted selection (which may do an inordinate amount of mere swapping of atoms from tree to tree in a manner more akin to point mutation rather than true crossover).

4 Experimental Results

This section describes some experiments in machine learning using hierarchical genetic algorithms. The author's computer program, consisting of 4341 lines of Common Lisp code, was run on a Texas Instruments Explorer II computer with a 25 megaHertz LISP microprocessor chip with 32 megabytes of internal memory and a half gigabyte of external hard disk memory. 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 randomized selections). Substantially similar results were obtained on several occasions for each experiment reported below.

4.1.1 Sequence Induction - Fibonacci Sequence

For this experiment, the problem is to induce the computational procedure (i.e. LISP S-expression) for the Fibonacci sequence. The environment in which adaptation is to take place consists of the first 20 elements of the actual Fibonacci sequence S = 1, 1, 2, 3, 5, 8, 13, 21, 34, 55,..., 4181, 6765. Recursion is known to be necessary to compute the Fibonacci sequence.

The set of functions available for this problem is F = {+, -, σ, *} and the set of atoms available is A = {0, 1, J}. For our purposes here we can view each atom as a function that requires no arguments in order to be evaluated. Thus, the combined set of functions and atoms is C = {+, -, σ, *, 0, 1, J} having 2, 2, 2, 2, 0, 0, and 0 arguments, respectively. In order to make the experiment more realistic, extraneous functions or atoms are included in all the experiments reported herein. The multiplication function here is

extraneous to a parsimonious solution of this problem. A population of 300 individuals was used. The algorithm begins by randomly generating 300 LISP S-expressions recursively using the items from set C. Examples of such random S-expressions included (+ J J), (* 0 (- J 1)), and (* (- (+ J 1) 0) (σ J)).

The raw fitness of an individual LISP S-expression in the population at any generational time step t is $\Sigma | P_{hj}(t) - S_j |$ where S_j is the actual Fibonacci sequence element and $P_{hj}(t)$ is the value returned by S-expression h for sequence position j. In this case, the smaller the raw fitness, the closer the match between the performance of the LISP S-expression involved and the actual Fibonacci sequence. Note that genetic algorithms do not require knowing any ultimate target solution or computing any differences between current trials and such an ultimate target solution. Genetic algorithms do use the relative performance of one individual compared to alternatives in the current population.

The best S-expression for generation 0 (the initial random population) was (σ (- J (σ (- J J) 0)) 0) with a raw fitness of 6765. The worst individual had a raw fitness of 28979. The average value of raw fitness was 17621. An adjusted fitness value $a_h = 1/(1+r)$ is then computed from the raw fitness r for each individual h. A

normalized fitness value $u_h = a_h / \Sigma a_h$ (ranging between 0 and 1 for each individual) is then computed for each individual. The average value of adjusted fitness for generation 0 was .0001 and the average normalized fitness was .0086. The number of exact matches for the best individual was 1 (out of 20). These predictably poor values for generation 0 serve as a useful baseline for the entire process.

A new population is then created from the current population. This process begins with the selection of a mating pool equal in size to the entire population using fitness proportionate reproduction (with replacement). In this run and each of the runs reported herein, the number of individuals involved with crossover equals 100% of the population for each generation. When these operations are completed, the new population replaces the old population.

The value of average fitness improved (i.e. dropped) from 17621 for generation 0 to 16969 and 15515 for generations 1 and 2, respectively. It then continued to improve monotonically to 5928 for generation 10. Between generations 11 and 24, the average fitness oscillated in the general neighborhood of 6000. Then, for generation 25, the value of average fitness improved to 5390. In addition, there was a monotonically improving trend for the fitness of the best

individual in the population from generation to generation. The worst individual in the population exhibited considerable variability (as is typical) but did improve overall. The average normalized fitness for each generation was very small until generation 16 (when an almost perfect individual appeared) and thereafter showed a substantial upwards movement.

The number of exact matches for the best individual of each generation started at 1 for generation 0, remained at 1 between generations 1 and 6, dropped to 0 at generation 7, rose to 2 between generations 8 and 13, rose to 18 for generations 14 and 15, rose to 19 for generations 16 through 21. Starting at generation 22, the best individual had a perfect score of 20 matches, namely (- (+ (σ (+ (- 0 1) J) 1) (σ (+ (- (- 0 1) 1) J) 0)) 0). This S-expression equals (+ (σ (- J 1) 1) (σ (- J 2) 0))).

The computer program takes approximately 150 seconds for 300 individuals for 26 generations. The process includes extensive interactive output consisting of two full-color graphs (with mouse-sensitive graph points for inspecting the various features) and five other windows for monitoring and controlling the process.

An asexual mutation operator which inserts a randomly generated sub-tree at a

randomly selected point was also programmed and tested in numerous runs. No run using only mutation and fitness proportionate reproduction produced a solution or exhibited any meaningful increase in population fitness. Moreover, an examination of the hereditary history (i.e. LISP audit trail indicating parents, crossover points, mutations points, etc.) of solutions achieved in various runs using crossover revealed that the solution never came about as a result of the mutation operation. When a point mutation operation was programmed and tested, it yielded similar negative results.

4.1.2 Sequence Induction - Cubic Polynomial Sequence

For this experiment, the problem was to induce the computational procedure for cubic polynomials such as $1+2J+J^2+J^3$. Note that neither the order of the polynomial required nor the size and shape of the computational procedures needed to solve this (and other problems herein) is provided to the problem solver in advance. The same functions and atoms as the Fibonacci sequence were used. Population size was 500. Starting with generation 5, a computational procedure emerged that returned values that exactly matched the actual cubic polynomial for all sequence positions. Similar results were obtained for a variety of different polynomials. Interestingly, in one run, the program unexpectedly factored the polynomial into

a product of factors $(J - r_k)$, where the r_k were the roots of the polynomial.

4.2.1 Automatic Programming - Pairs of Linear Equations

The problem of automatic programming requires developing a computer program that can produce a desired output for a given set of inputs. 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 two real-valued variables. The environment consisted of a suite of 10 pairs of equations (to avoid 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 atoms is A = {A11, A12, A21, A22, B1, B2}. The raw fitness of a particular S-expression is the sum of the Euclidian distances between the known solution point in the plane and the point produced by the S-expression for all 10 pairs of equations in the test suite.

Population size was 300. 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)) appearing 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)). Between generations 27 and 30, the average normalized fitness rises to .39 (as the perfect solution starts dominating).

4.2.2 Automatic Programming - 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 square root function S [which returns a LISP complex number, e.g. (S -4) is #C(0, 2)], and a modified division operation % (which returns a value of zero for division by zero). A population of size 300 was used for 3l

generations. The environment consisted of a suite of 10 quadratic equations (with some purely real roots, some purely imaginary roots, and some complex-valued roots). A correct solution to the problem emerged at generation 22, namely, the S-expression (- (S (- (* (% B 2) (% B 2)) C)) (% B 2)).

4.2.3 Automatic Programming - Trigonometric Identities

For this group of experiments, the problem was to derive various trigonometric identities. This particular group of experiments yielded a number of unexpected results. The environment consisted of a Monte Carlo suite of 20 pairs of randomly generated X values between 0 and 2Π radians and the value of cox 2X (which is equivalent to 1 - 2 $\sin^2$ X). The available functions were SIN, multiplication, and subtraction (with the addition and cosine function were intentionally deleted from the repertoire of available functions). 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)) was obtained after 16 generations. In one run with cos 2X, the S-expression (SIN (- (- 2 (* X 2)) (SIN (SIN (SIN (SIN (SIN (SIN (* (SIN (SIN 1)) (SIN (SIN 1)))))))))))), where 1 is in radians, was obtained as the best

individual. This expression approximately equals sin (Π/2 - 2X).

4.3.1 Machine Learning - Boolean Multiplexer Function

For this experiment, the problem is to find the Boolean expression which gives the correct Boolean output value for a given Boolean multiplexer function. 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 6-multiplexer (where k = 2), if the two address bits a_1a_0 are 11, then the output is the third data bit d_3. The Boolean multiplexer function can be represented in disjunctive normal form as (OR (AND A1 A0 D3) (AND A1 (NOT A0) D2) (AND (NOT A1) A0 D1) (AND (NOT A1) (NOT A0) D0). This function has been studied in connection with neural nets (Barto et. al. 1985) and classifier systems (Wilson 1987a)

The combined set of functions and atoms for this problem is C = {NOT, OR, OR, OR, AND, AND, IF, IF, A0, A1, D0, D1, D2, D3 } with 1,2,3,4,2,3,2,3,0,0,0,0,0, and 0 arguments, respectively. Note that the OR,

AND, and IF functions appear with varying number of arguments (e.g. 2, 3, or 4). For example, the IF function with 3 arguments is an if-then-else function. Population size was 300. The environment consisted of the 2^w (where w = k+2^k) possible inputs.

Initial random individuals include contradictions such as (AND A0 (NOT A0)), inefficiencies such as (OR D3 D3), irrelevancies such as (IF A0 A0 (NOT A1)), and nonsense such as (IF (IF (IF D2 D2) D2) D2). The best individual from generation 0 was (IF A0 D1 D2) with a raw fitness value of 16 (i.e. 16 mismatches out of a possible 64). This individual uses just one of the address bits (A0) to decide whether the output is data line D1 or D2 and can never give an output of D0 or D3. Nonetheless, in the valley of the blind, the one-eyed man is king.

The average raw fitness of the population immediately begins improving from the baseline value for generation 0 of 29.05 to 26.89, 25.74, 23.78, 22.09, 21.38, 20.13, 19.91, etc. In generation 9 a best individual arises that has only 12 mismatches, namely (IF (IF A0 (OR A1 D0)) D3 (IF A0 D1 D2)). Note that (IF A0 D1 D2) from generation 0 is now embedded as a sub-expression within this new individual. In generation 11, a new best individual arises that has only 8 mismatches, namely (IF A0 D1 (IF Al D2 D0)). The sub-expression (IF Al D2 D0) contributes substantially to this improved performance because it perfectly deals with the case

when A0 is NIL (False) by taking either data line D2 or D0 as its output (depending on A1). Note also that (IF A0 D1 ...) is partially correct when A0 is T (True). In generations 12, 13, and 14, a new individual arises with only 4 mismatches, namely, (IF (IF (A0 (OR A1 D0) D3 (IF A0 D1 (IF Al D2 D0)))).

In generation 15, a perfect solution i.e. an individual with 0 mismatches) emerges, namely, (IF (IF A0 A1) D3 (IF A0 D1 (IF Al D2 D0)) as a result of a crossover where the unfit sub-expression (IF (A0 (OR Al D0))) is replaced by the more fit sub-expression (IF A0 A1).

The interpretation of this solution expression is as follows: The output of the multiplexer is D3 if (IF A0 A1) is true (i.e. the two address bits are 11). Note that IF function in LISP (unlike the predicate calculus) is equivalent to the AND function. If that is not true, the output is D1 if A0 is true (because the two address bits are necessarily now 01). Note that setting the output to D1 if merely A0 were true in a vacuum is not a correct solution to the problem. However, after (IF A0 Al) has been considered (and found to be false), then (IF A0 D1 ...) is correct. Finally, (IF Al D2 D0) now handles the case when address bit A0 must necessarily be NIL. In this context, the partially correct sub-expression that was around since generation 0, namely (IF A1 D2 D0), sets the output of the multiplexer to D2 if A1 is T (because the two address

bits are 10) and, otherwise, it sets the output to D0 (because the two address bits are 00).

Note that a default hierarchy emerged here which incorporated partially correct sub-rules into a perfect overall procedure by dealing with ever more specific cases. Although default hierarchies are considered desirable in classifier systems (Goldberg 1989, Holland 1986), none emerged in Wilson's (1987) otherwise noteworthy experiments involving classifier systems and the multiplexer.

The perfect solution above arose after processing 4500 individuals. Others have required processing as few as 3900 individuals. Note that the hierarchical algorithm does not start with any advance information identifiying inputs versus outputs or any advance information about the size and shape of the ultimate solution.

4.3.2 Machine Learning - The Parity Function

For this experiment, the problem is to find the Boolean expression for the Boolean parity function. The k-parity function takes k Boolean arguments and returns T if an odd number of its arguments are T and returns NIL otherwise. The exclusive-or function and the k-parity function were not realizable by early simple perceptrons (Minsky and Papert

1969) and are, as a result, commonly used as test functions for multi-layered non-linear neural networks (Rumelhart <u>et</u>. <u>al</u>. 1986). Moreover, these functions yield uninformative schema (similarity templates) with conventional linear genetic algorithms using fixed length binary strings so that these functions are not realizable with such linear genetic algorithms.

The combined set of functions and atoms used for the 3-parity function was C = {AND, OR, NOT, IF, D2, D1, D0} with 2, 2, 1, 3, 0, 0, and 0 arguments, respectively. Population size was 300. The S-expression (AND (IF D2 D0 (NOT D0)) D1) appeared in generation 0 and was correct 6 out of the 8 cases constituting the environment. In generation 4, a rather complex S-expression appeared which contained part of this individual from generation 0 and was correct 7 out of 8 times. Finally, in generation 5, a new individual emerged which was correct in all 8 cases, namely (IF (IF D2 D0 (NOT D0)) D1 (NOT D1)). Note that this final individual consisted of a substantial portion of the earlier best individual. Note also that the sub-expression (IF (D2 D0 (NOT D0))) is a partially correct solution to the problem (i.e. if only the two items of data D0 and D2 need to be considered) and that this sub-expression is embedded in a default hierarchy using it in conjunction with the value of D1 to produce the overall correct solution to the problem.

The exclusive-or function (i.e. parity function of order 2) was similarly discovered and then successfully used in learning parity functions of up to order 10.

4.4 Planning

Nilsson (1988a) has presented a robotic action network that solves a 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 experiment here, the goal is to automatically generate a general plan that solves this problem using hierarchical genetic algorithms.

Three lists are involved in the formulation of the problem. The GOAL-LIST is the list specifying the desired final order in which the blocks are to be stacked in the target tower (i.e. "FRUITCAKE" or "UNIVERSAL"). The STACK is the list of blocks that are currently in the target tower (where the order is important). The TABLE is the list of blocks that are currently not in the target tower. 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.

The environment can be viewed as consisting of up to (N+1)! different initial configurations of N blocks in the STACK list and on the TABLE list. The raw fitness of a particular individual plan in the population is the number of initial configurations for which the particular plan produces the desired final configuration of blocks after the plan is executed. The computation of fitness can be significantly shortened by consolidating functionally equivalent initial configurations.

In the problem as stated, three sensors dynamically track the environment in the formulation of the problem. TB is a sensor that 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. NN is a sensor that dynamically specifies the next needed block for the STACK (i.e. the immediate predecessor of TB in GOAL-LIST). CS dynamically specifies the CAR of the STACK (i.e. the top block). Thus, the set of atoms available for solving the problem here is A = {TB, NN, CS}. Each of these atoms may assume the value of one of the block labels or the value NIL.

The set of functions available for solving the problem here contains 6 functions F = {MS, MT, DU, QUOTE, NOT, EQ}. The function

MS has one argument and moves block X to the top of the STACK if X is on the table. The function MT has one argument and moves the top item to the TABLE if the STACK contains X anywhere in the STACK. The iterative function DU ("do until") has two arguments, namely a predicate PRED and some WORK. Both the MS and MT functions have return values, although their true functionality consists of their side effects on STACK and TABLE. The function DU tests the predicate PRED and does the WORK (via the LISP evaluation function EVAL) repeatedly until the predicate PRED becomes T (True). Note that the fact that each function returns some value (in addition to whatever side effects it has on the STACK and TABLE) and the flexibility of the LISP language guarantees that the DU function can be executed and evaluated for any combination of functions and arguments (however unusual, pointless, or counter-productive). Since individuals in the population will often contain complicated nestings of DU functions and unsatisfiable termination predicates, limits are placed on both the number of iterations allowed (without preventing any plan from being executed and evaluated). The LISP function QUOTE has one argument and suppresses the usual immediate evaluation of arguments that occurs in LISP and thereby provides a way to prevent premature evaluation of the WORK argument of a DU function until it is inside the function DU. Note that the QUOTE function also has the interesting and highly

epistatic effect of smothering the functionality of its arguments when it appears elsewhere.

A population of 300 individual plans was used. The initial random population of plans had predictably low fitness. Typical random 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 the performs the useless Boolean comparison on the return value of the MT function with the sensor value NN. The second plan futilely attempts to move the block TB (which already is in the STACK) from the TABLE to the STACK. The single best individual in this initial population of plans typically can successfully handle perhaps one or two of the very simplest one or two initial configurations.

After about 5 generations, we typically see the emergence of perhaps one plan in the population that correctly deals with the simplest group of cases in the environment (i.e. the cases in which the blocks, if any, in the initial STACK are already all in the correct order and in which there are no out-of-order blocks on top of those blocks). In several runs, the rather parsimonious (DU (QUOTE (MS NN)) (NOT NN)) emerged as a partially correct plan. This plan works by improving a partially correct initial STACK by moving needed blocks (NN) in the correct sequence from the TABLE onto the STACK until there are no more blocks

needed to finish the STACK (i.e. the sensor NN is no longer a block).

After about 10 generations, the best single individual in the population is typically a plan that achieves a perfect score (that is, the plan produces the final desired configuration of blocks in the STACK for all initial configuration of blocks in the environment). One such plan is (NOT (EQ (DU (QUOTE (MT CS)) (NOT NN)) (EQ (MS (DU (QUOTE (MS NN)) (NOT NN))) (DU NN (QUOTE TB))))). Note that this plan contains a default hierarchy. In particular, the sub-plan (DU (QUOTE (MS NN)) (NOT NN)) comes from an ancestor from an earlier generation (which performed correctly for a simple set of cases of initial configurations). This sub-plan is now incorporated as a sub-plan (i.e. a small "building block"). Note also that another sub-plan (DU (QUOTE (MT CS)) (NOT NN)) from another individual from an earlier generation correctly deals with the remaining cases by first moving out-of-order blocks from the STACK to the TABLE until the STACK contains no incorrect blocks. By combining these two somewhat fit sub-plans from earlier generations, a solution to the entire problem is achieved. Note also that the third sub-plan, namely (DU NN (QUOTE TB)), and the functions NOT and EQ perform no useful function (but also do no harm).

4.5 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 connection with the claim that connectionist neural networks cannot possibly solve this type of problem. In the simplified experiment here, the retina has 6 pixels (with wrap-around) and the shape consists of three consecutive binary 1's.

The functions available are a zero-sensing function H0, a one-sensing function H1, ordinary multiplication, and a disjunctive function U. The atoms available are the integers 0, 1, and 2, and a universally quantified atom k.

LISP's comparative tolerance as to typing is well suited to pattern recognition problems where it is desirable to freely combine numerical concepts such as positional location (either absolute, or universally quantified), relative displacement (e.g. the symbol 2 pixels to the right) with various combinations of Boolean tests. The functions U and * so defined resolve potential type problems that would otherwise arise when integers identify positions in the retina.

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 l'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).

5 Robustness

The existence and nurturing of a population of disjunctive alternative solutions to a problem allows hierarchical genetic algorithms to effectively perform even when the environment changes. To demonstrate this ability, 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 $x^2 + 2x + 1$. Population size was 300. 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.

6 Theoretical Discussion

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 schemata with above-average (below-average) fitness appear in succeeding generations at an approximately exponentially increasing (decreasing) rates. Holland also showed that the 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 mathematical near optimal. This allocation of trials is most nearly optimal when ε is small. ε is the defining length $\delta(H)$ of the schemata

involved (i.e. the distance between the outermost specific, non-* symbols) divided by L-1 (i.e. the number of points where crossover may occur). Therefore, ε is short when $\delta(H)$ is short (i.e. the schemata is a small, short, compact "building block"). Thus, genetic algorithms process short-defining length schemata most favorably and problems structured so that their solutions can be "built up" from such small "building blocks" are most optimally handled by genetic algorithms.

In hierarchical genetic algorithms, the individuals in the population are LISP S-expressions (i.e. rooted point-labeled trees in a plane), instead of linear character strings. The set of similar individuals sharing common features (i.e. the schemata) is a hyperspace of LISP S-expressions (i.e. rooted point-labeled trees in a plane) 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 on optimal allocation or trials and Holland's result on growth (or decay) of number of occurrences of schemata as a result of fitness proportionate reproduction alone (1975) do not depend, in any way, on the character of the individual objects in the population. The deviation from this optimal rate of growth (or decay) of schema is caused by the crossover operation. This deviation is relatively small when the number of points defining the common feature (i.e. the number of points in the sub-tree) is relatively small. 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 the schemata are coextensive with a 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.

Hierarchical genetic algorithms are 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 nucleiotide 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 schemata fitness for each schemata represented by that individual. In contrast, the proactive computational procedure carried out by a LISP S-expression in a hierarchical genetic algorithm 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 schemata fitness for each schemata represented by that individual. That is, hierarchical genetic algorithms employ 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).

7 Conclusions

The examples from the five areas of artificial intelligence, including sequence induction, automatic programming, function learning, robotic planning, and pattern recognition support the view that computational procedures (i.e. computer programs, LISP S-expressions) can be built up from appropriate small "building blocks" using hierarhical genetic algorithms.

Acknowledgments

Dr. Thomas Westerdale of Birkbeck College at the University of London, Dr. Martin A. Keane of Third Millennium Venture Capital Limited in Chicago, and John Perry of Texas Instruments Inc. in San Francisco made valuable comments on drafts of this paper. Eric Mielke of the Texas Instruments Education Center in Austin significantly improved execution time of the author's crossover operation.

References

[Barto et. al. 1985] 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.

[Cramer 1985] 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 and Steenstrup 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.

[De Jong 1987] 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.

[Fogel et. al. 1966] Fogel, L. J., Owens, A. J. and Walsh, M. J. Artificial Intelligence through Simulated Evolution. New York: John Wiley 1966.

{Fujuki 1986] Fujuki, 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, 1986.

[Goldberg 1989] Goldberg, David E. Genetic Algorithms in Search, Optimization, and Machine Learning. Reading, MA: Addison-Wesley 1989.

[Hicklin 1986] 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.

[Hinton 1988] Hinton, Geoffrey, Neural Networks for Artificial Intelligence. Santa Monica, CA: Technology Transfer Institute. Documentation dated December 12, 1988.

[Holland 1975] Holland, John H. Adaptation in Natural and Artificial Systems, Ann Arbor, MI: University of Michigan Press 1975.

[Holland 1986] 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.

[Lenat 1983] 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.

[Minsky and Papert 1969] Minsky, Marvin L. and Papert, Seymour A. Perceptrons. Cambridge, MA: The MIT Press. 1969.

[Nilsson 1988a] Nilsson, Nils J. Action networks. Draft Stanford Computer Science Department Working Paper, October 24, 1988. Stanford, CA: Stanford University. 1988.

[Nilsson 1988b] Nilsson, Nils J. Private Communication. 1988.

[Rumelhart et. al. 1986] 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.

[Schaffer 1987] 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 1987.

[Smith 1980] Smith, Steven F. A Learning System Based on Genetic Adaptive Algorithms. PhD dissertation. University of Pittsburgh 1980.

[Wilson 1987] Wilson, S. W. Classifier Systems and the animat problem. Machine Learning, 3(2), 199-228, 1987.