A Hierarchical Approach to Learning the Boolean Multiplexer Function

John R. Koza
Computer Science Department
Stanford University
Stanford, CA 94305 USA
Koza@Sunburn.Stanford.Edu
415-941-0336

ABSTRACT

This paper describes the recently developed genetic programming paradigm which genetically breeds populations of computer programs to solve problems. In genetic programming, the individuals in the population are hierarchical compositions of functions and arguments. Each of these individual computer programs is evaluated for its fitness in handling the problem environment. The size and shape of the computer program needed to solve the problem is not predetermined by the user, but instead emerges from the simulated evolutionary process driven by fitness. In this paper, the operation of the genetic programming paradigm is illustrated with the problem of learning the Boolean 11-multiplexer function.

1. Introduction and Overview

We start by reviewing previous work in the field of genetic algorithms and previous work in the field of induction of computer programs. We then describe the recently developed genetic programming paradigm and apply it to the problem of learning the Boolean 11-multiplexer problem.

2. Background

John Holland of the University of Michigan presented the pioneering formulation of genetic algorithms for fixed-length character strings in *Adaptation in Natural and Artificial Systems* (Holland 1975). The genetic algorithm is a highly parallel mathematical algorithm that transforms a population of individual mathematical objects (typically fixed-length binary character strings) into a new population using operations patterned after natural genetic operations such as sexual recombination (crossover) and fitness proportionate reproduction (Darwinian survival of the fittest). The genetic algorithm begins with an initial population of individuals (typically randomly generated). It then iteratively evaluates the individuals in the population for fitness with respect to the problem environment and produces a new population by performing genetic operations on individuals selected from the population with a probability proportional to fitness.

Representation is a key issue in genetic algorithm work because genetic algorithms directly manipulate the coded representation of the problem and because the representation scheme can severely limit the window by which the system observes the world. Fixed length character strings present difficulties in problems where the desired solution is hierarchical and where the size and shape of the solution is unknown in advance. The need for more powerful representations has been long recognized (De Jong 1988).

The structure of the individual mathematical objects that are manipulated by a genetic algorithm can be more complex than the fixed length character strings. Smith (1980) departed from the early fixed-length character strings by introducing variable length strings, including strings whose elements were if-then rules (rather than single characters).

Holland's introduction of the classifier system (1986) continued the trend towards increasing the complexity of the structures undergoing adaptation. The classifier system is a cognitive architecture into which the genetic algorithm is embedded so as to allow adaptive modification of a population of if-then rules (whose condition and action parts are fixed length binary strings) using a bucket brigade algorithm for allocation of credit.

Wilson (1987b) introduced hierarchical credit allocation into Holland’s bucket brigade algorithm in to encourage the creation of hierarchies of rules in lieu of the exceedingly long sequences of rules that are otherwise characteristic of classifier systems. Wilson's efforts recognize the central importance of hierarchies in representing the tasks and subtasks (i.e. programs and subroutines) that are needed to solve complex problems.

Goldberg et. al (1989) introduced the messy genetic algorithm (mGA) which processes populations of variable length character strings. Messy genetic algorithms solve problems by combining relatively short, well-tested sub-strings that deal with part of a problem to form longer, more complex strings that deal with all aspects of the problem.

3. Background on Genetic Programming Paradigm

The recently developed genetic programming paradigm is a method of program induction which genetically breeds a population of computer programs to solve problems.

Work in program induction goes back to the 1950's. Friedberg's early work (1958, 1959) attempted to artificially generate entire computer programs to solve problems. Friedberg used a rather ineffective form of search to generate computer programs in a hypothetical assembly language for a hypothetical computer with a one-bit register.

Cramer (1985) applied genetic algorithms to program induction. Cramer used the genetic algorithm operating on fixed length character strings to generate computer programs with a fixed structure and reported on the difficult and highly epistatic nature of the problem.

Fujiki and Dickinson (1987) implemented analogs of the genetic operations from the conventional genetic algorithm to manipulate the individual if-then clauses of a LISP computer program consisting of a single conditional (COND) statement. The individual if-then clauses of Fujiki and Dickinsons' COND statement were parts of a strategy for playing the iterated prisoner’s dilemma game.

In the recently developed genetic programming paradigm, the individuals in the population are compositions of functions and terminals appropriate to the particular problem domain. The set of functions used typically includes arithmetic operations, mathematical functions, conditional logical operations, and domain-specific functions. The set of functions must be closed in the sense that each function in the function set must be defined for any combination of elements from the range of every function that it may encounter and every terminal that it may encounter. The set of terminals used typically includes inputs (sensors) appropriate to the problem domain and constants. The search space is the hyperspace of all possible compositions of functions and terminals that can be recursively composed using the available functions and terminals.

The symbolic expressions (S-expressions) of the LISP programming language provide an especially convenient way to create and manipulate hierarchical compositions of functions and terminals. S-expressions in LISP correspond directly to the parse tree that is internally created by most compilers at the time of compilation. The parse tree is nothing more than a direct mapping of the given composition of functions (i.e. the given computer program). We need access to this parse tree to do crossover on the parts of computer programs. The LISP programming language gives us this convenient access to the parse tree, the ability to conveniently manipulate this program as if it were data, and the convenient ability to immediately execute a newly created parse tree.

The genetic programming paradigm genetically breeds computer programs to solve problems by executing the following two steps:

(1) Generate an initial population of random compositions (computer programs) of the functions and terminals of the problem.

(2) Iteratively perform the following until the termination criterion has been satisfied:

 (a) Execute each program in the population and assign it a fitness value according to how well it solves the problem.

 (b) Create a new population of computer programs by applying the following two primary operations:

  (i) Allow existing computer programs to survive with a Darwinian probability based on their fitness.

  (ii) Create new computer programs by genetically recombining randomly chosen parts of two existing programs (each chosen from the population with a Darwinian probability based on fitness).

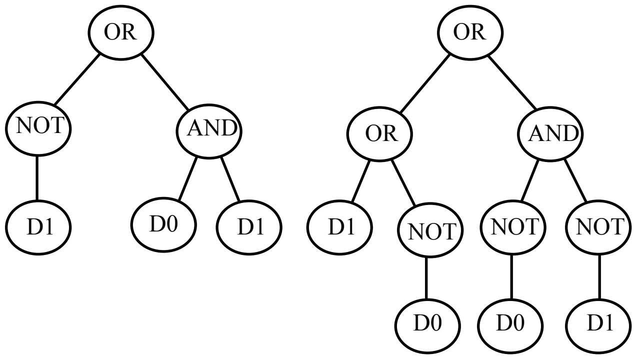
The basic genetic operations for the genetic programming paradigm are fitness proportionate reproduction and crossover (recombination). Fitness proportionate reproduction is the basic engine of Darwinian reproduction and survival of the fittest. It operates here in the same way as it does for conventional genetic algorithm.

The crossover (recombination) operation for the genetic programming paradigm is a sexual operation that operates on two parental LISP S-expressions and produces two offspring S-expressions using parts of each parent. In particular, the crossover operation creates new offspring S-expressions by exchanging sub-trees (i.e. sub-lists) between the two parents. Because entire sub-trees (i.e. sub-lists) are swapped, this crossover operation always produces syntactically and semantically valid LISP S-expressions as offspring. For example, consider the two parental LISP S-expressions:

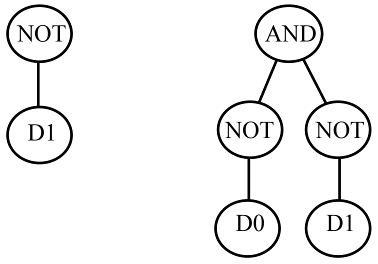
`(OR` **<u>`(NOT D1)`</u>** `(AND D0 D1))`

`(OR (OR D1 (NOT D0))` **<u>`(AND (NOT D0) (NOT D1)`</u>**`)`

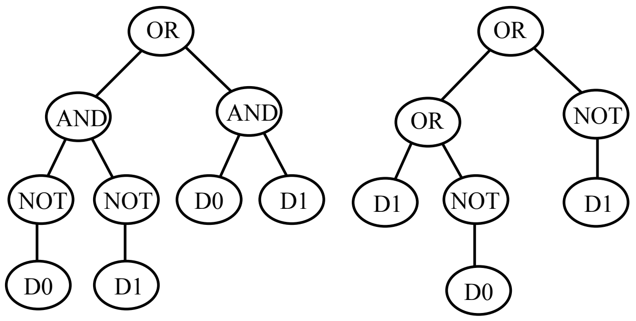
These two LISP S-expressions can be depicted graphically as rooted, point-labeled trees with ordered branches. The two parental LISP S-expressions are shown below:



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



These two crossover fragments correspond to the bold, underlined sub-expressions (sub-lists) in the two parental LISP S-expressions shown above. The two offspring resulting from crossover are shown below.



Note that the first offspring produced is an S-expression for the even parity function:

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

Although one might think that computer programs are so epistatic that they could only be genetically bred in a few especially congenial problem domains, we have shown that computer programs can be genetically bred to solve problems in a surprising variety of different areas of machine learning and artificial intelligence, including

- planning (e.g. navigating an artificial ant to find food along an irregular trail; developing a robotic action sequence that can stack blocks) [Koza 1990b],
- emergent behavior (e.g. discovering a computer program for locating food, carrying food to the nest, and dropping pheromones, which, when executed by all the ants in an ant colony, produces interesting higher level "emergent" behavior) [Koza 1991a],
- finding minimax strategies for games (e.g. differential pursuer-evader games; discrete games in extensive form) by both evolution and co-evolution [1991b],
- optimal control (e.g. centering a cart and balancing a broom in minimal time by applying a bang-bang force to the cart) (Koza and Keane 1990a, 1990b],
- discovering inverse kinematic equations (e.g. to move a robot arm to designated target points) [Koza 1991f],
- sequence induction (e.g. inducing a recursive procedure for generating sequences such as the Fibonacci and the Hofstadter sequences) [Koza 1989],
- symbolic "data to function" regression, integration, differentiation, and symbolic solution to general functional equations (including differential equations with initial conditions, integral equations, and inverse problems) [Koza 1990],
- empirical discovery (e.g. rediscovering Kepler's Third Law; rediscovering the well-known non-linear econometric "exchange equation" MV = PQ from actual, noisy time series data for the money supply, the velocity of money, the price level, and the gross national product of an economy) [Koza 1991d],
- automatic programming (e.g. solving pairs of linear equations, solving quadratic equations for complex roots, and discovering trigonometric identities),
- pattern recognition (e.g. translation-invariant recognition of a one-dimensional shape in a linear wrap-around retina),

- concept formation and decision tree induction [Koza 1991c],
- generation of random numbers (using entropy as fitness) [Koza 1991e], and
- simultaneous architectural design and training of neural nets [Koza and Rice 1991a].

A visualization of the application of the genetic programming paradigm to planning, emergent behavior, empirical discovery, inverse kinematics, game playing, and the Boolean 11-multiplexer problem discussed in this paper can be viewed in the *Artificial Life II Video Proceedings* videotape [Koza and Rice 1991b].

4. Boolean 11-Multiplexer Function

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

In this paper, the problem is to learn the Boolean 11-multiplexer function. In general, the input to the Boolean multiplexer function consists of k "address" bits a_i and 2^k "data" bits d_i and is a string of length $k+2^k$ of the form $a_{k-1}...a_1a_0$ $d_{2^k-1}...d_1$ d_0. The value of the multiplexer function is the value (0 or 1) of the particular data bit that is singled out by the k address bits of the multiplexer. For example, for the 11-multiplexer (where k = 3), if the three address bits $a_2a_1a_0$ are 110, then the multiplexer singles out data bit number 6 (i.e. d_6) to be its output. The Boolean multiplexer function with $k+2^k$ arguments is one of 2^{k+2^k} possible Boolean functions of $k+2^k$ arguments. Thus, the search space for the Boolean multiplexer is of size 2^{k+2^k}. When k=3, this search space is of size $2^{2^{11}}$ = 2^{2048} , which is approximately 10^{616}.

The solution of the Boolean 11-multiplexer problem (involving a search space of size 10^{616}) will serve to show the interplay, in the genetic programming paradigm, of

- the genetic variation inevitably created in the initial random generation,
- the small improvements for some individuals in the population via localized hill-climbing from generation to generation,
- the way particular individuals become specialized so as to be able to correctly handle certain sub-cases of the problem (i. e. case-splitting),
- the creative role of crossover in recombining valuable parts of more fit parents to produce new individuals with new capabilities, and
- how the nurturing of a large population of alternative solutions to the problem (rather than a single point in the solution space) helps avoid false peaks in the search for the solution to the problem.

This problem will also serve to illustrate the importance of hierarchies in solving problems and making the ultimate solution understandable. Moreover, the progressively changing size and shape of the various individuals in the population in various generations shows the flexibility resulting from not determining the size and shape of ultimate solution or the intermediate results in advance

The five major steps in setting up the genetic programming paradigm require determining: (1) the set of terminals, (2) the set of functions, (3) the fitness function, (4) the parameters for the run, and (5) the criterion for designating a result and terminating a run.

The first step in applying the genetic programming paradigm to a problem is to select the set of terminals that will be available to the algorithm for constructing the computer programs (LISP S-expressions) that will try to solve the problem. For some problems (in particular, Boolean function learning problems), this choice is especially straight-forward and obvious. The set of terminals for this problem consists of the 11 inputs to the Boolean 11-multiplexer. The algorithm cannot distinguish as to whether the terminals are address lines or data lines. The terminal set for this problem is

$$T = \{A0, A1, A2, D0, D1, \dots, D7\}.$$

The second step in applying the genetic programming paradigm to a problem is to select the set of functions. The set of functions for this problem is

$$F = \{AND, OR, NOT, IF\}.$$

This set of basic logical functions satisfies the closure property. This set is sufficient to realize any Boolean function. In addition, this set is a convenient set in that it often produces easily understood S-expressions.

The third step in applying the genetic programming paradigm to a problem is to determine the fitness function. Fitness is often evaluated over fitness (environmental) cases. The fitness cases here consist of the 2^{11} possible combinations of the 11 arguments along with the associated correct value of the 11-multiplexer function. For the 11-multiplexer (where $k = 3$), there are 2048 such combinations of arguments. In this particular paper, we use the entire set of 2048 combinations of arguments (i.e. we do not use sampling); however, sampling is an obvious option. The standardized fitness is the sum, taken over all 2048 fitness cases, of the Hamming distances between the Boolean value returned by the S-expression for a given combination of arguments and the correct Boolean value. This fitness measure is equivalent to the number of mismatches (i.e. sum of errors).

The fourth major step in using the genetic programming paradigm is selecting the values of certain parameters for running the algorithm. Population size is the most important parameter. It is 4000 here. Each new generation is created from the preceding generation by applying fitness proportionate reproduction to 10% of the population and by applying crossover to 90% of the population (with one parent selected proportionate to fitness). In selecting crossover points, 90% were internal (function) points of the tree and 10% were external (terminal) points of the tree. Mutation was not used. For the practical reason of conserving computer time, the depth of initial random S-expressions was limited to 5 and the depth of S-expressions created by crossover was limited to 20.

Finally, the fifth major step in using the genetic programming paradigm is the criterion for terminating a run and designating a result. We terminate a given run when either (1) the genetic programming paradigm produces a computer program whose standardized fitness attains the perfect score of zero, or (2) 51 generations (i.e. the initial random generation and 50 others) have been run.

Note that the first two of these five major steps in applying the genetic programming paradigm corresponds to the step (performed by the user) of determining the representation scheme in the conventional genetic algorithm operating on character strings (that is, determining the chromosome length, alphabet size, and the mapping between the problem and chromosomes). The last three of the these five steps apply to both methodologies.

In addition, note that the step (performed by the user) of determining the set of primitive functions in the genetic programming paradigm is equivalent to a similar step in other machine learning paradigms. For example, this same determination of primitive functions occurs in the induction of decision trees using ID3 (Quinlan 1986, Koza 1991c) when the user selects the functions that can appear at the internal points of the decision tree. Similarly, this same determination occurs in neural net problems when the user selects the external functions that are to be activated by the output of a neural network. The same user determination occurs in other machine learning paradigms (although the name given to this omnipresent determination varies and is often considered by the researcher to be implicit in the statement of his or her problem).

We illustrate the overall process by discussing one particular run of the Boolean 11-multiplexer in detail. Later, we will present statistics involving multiple runs.

The process begins with the generation of the initial random population (i.e. generation 0). Predictably, the initial random population includes a variety of highly unfit individuals. Many individual S-expressions in this initial random population are merely constants, such as the contradictory `(AND A0 (NOT A0))`. Other individuals are passive and merely pass a single input through as the output, such as `(NOT (NOT A1))`. Other individuals inefficiently do the same, such as `(OR D7 D7)`. Some initial random individuals, such as `(IF D0 A0 A2)`, make a decision based on precisely the wrong argument (i.e. using a data bit, rather than address bits, to select the output). Many initial random individuals are partially blind in that they do not incorporate all 11 arguments that are necessary to solve the problem. Some initial random S-expressions are just nonsense, such as

```
(IF (IF (IF D2 D2 D2) D2 D2) D2 D2).
```

Nonetheless, even in this highly unfit initial random population, some individuals are somewhat more fit than others. For this particular run, the individuals in the initial random population had values of standardized fitness ranging from 768 mismatches (i.e. 1280 matches or hits) to 1280 mismatches (i.e. 768 matches). As it happens, a total of 23 individuals out of the 4000 in this initial random population tied with the best score of 1280 matches (i.e. 768 mismatches) on generation 0. One of these 23 best scoring individuals from generation 0 was the S-expression

```
(IF A0 D1 D2).
```

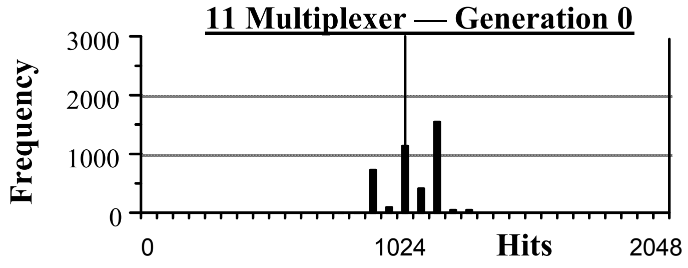
This individual has obvious shortcomings. Notably, this individual is partially blind in that it uses only three of the 11 necessary terminals of the problem. As a consequence of this fact alone, this individual cannot possibly be a correct solution to the problem. This individual nonetheless does some things right. For example, it uses one of the three address bits (A0) as the basis for its action. It could easily have done this incorrectly and used one of the eight data bits. In addition, this individual uses only data bits (D1 and D2) as its output. It could have used address bits. Moreover, if A0 (which is the low order binary bit of the 3-bit address) is T (True), this individual selects one of the three odd numbered data bits (D1) as it output. Moreover, if A0 is NIL, this individual selects one of the three even numbered data bits (D2) as its output. In other words, this individual correctly links the parity of the low order address bit A0 with the parity of the data bit it selects as its output. This individual is far from perfect, but it is far from being without merit. It is more fit than 3977 of the 4000 individuals in the population. In the valley of the blind, the one-eyed man is king.

In contrast, the worst individual in the population for the initial random generation had a standardized fitness of 1280 (i.e. only 768 matches). It is shown below:

```
(OR (NOT A1) (NOT (IF (AND A2 A0) D7 D3))).
```

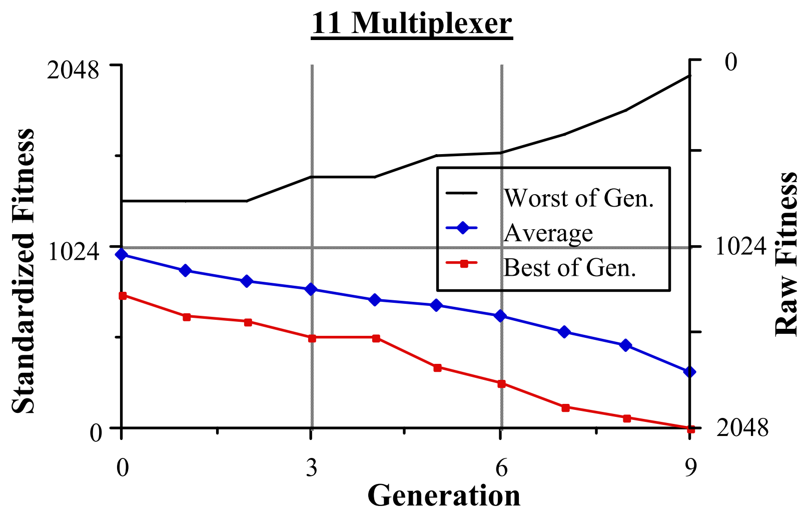
The average standardized fitness for all 4000 individuals in the population for generation 0 is 985.4. This value of average standardized fitness for the initial random population forms the baseline and serves as a useful benchmark for monitoring later improvements in the average standardized fitness of the population as a whole.

The hits histogram is a useful monitoring tool based on the number of hits (i.e. matches). This histogram provides a way of viewing the population as a whole for a particular generation. The horizontal axis of the hits histogram is the number of hits (i.e. matches) and the vertical axis is the number of individuals in the population scoring that number of hits. Fifty different levels of fitness are represented in the hits histogram for the population at generation 0 of this problem. In order to make this histogram legible for this problem, we have divided the horizontal axis into buckets of size 64. For example, 1553 individuals out of 4000 (i.e. about 39%) had between 1152 and 1215 matches (hits). The mode (i.e. highest point) of the distribution occurs at 1152 matches (hits). The figure below shows the hits histogram of the population for generation 0.



A new population is then created from the current population using the operations of Darwinian fitness proportionate reproduction and crossover. When these operations are completed, the new population (i.e. the new generation) replaces the old population. In going from the initial random generation (generation 0) to generation 1, the genetic programming paradigm works with the inevitable genetic variation existing in an initial random population. The initial random generation is an exercise in blind random search. The search is a parallel search of the search space over the 4000 individual points .

The figure below shows the standardized fitness (i.e. mismatches) for generations 0 through 9 of this run for the best single individual in the population, the worst single individual in the population, and the average for the population.



The average standardized fitness of the population immediately begins improving (i.e. decreasing) from the baseline value of 985.4 for generation 0 to about 891.9 for generation 1. As it happens, in this particular run, the average standardized fitness improves (i.e. decreases) monotonically between generation 2 and generation 9 and assumes values of 845, 823, 763, 731, 651, 558, 459, and 382, respectively. We usually see a generally improving trend in average standardized fitness from generation to generation, but not necessarily a monotonic improvement.

In addition, we similarly usually see a generally improving trend in the standardized fitness of the best single individual in the population from generation to generation. As it happens, in this particular run of this particular problem, the standardized fitness of the best single individual in the population improves (i.e. decreases) monotonically between generation 2 and generation 9. In particular, it assumes values of 640, 576, 384, 384, 256, 256, 128, and 0 (i.e. a perfect score), respectively.

On the other hand, the standardized fitness of the worst single individual in the population fluctuates considerably. For this particular run, the standardized fitness of the worst individual starts at 1280, fluctuates considerably between generations 1 and 9, and then deteriorates (increases) to 1792 by generation 9.

The standardized fitness for the best single individual in the population improved (i.e. dropped) to 640 mismatches (i.e. 1408 matches) for generations 1 and 2 of the run. Only one individual in the population attained this best score in generation 1, namely

```
(IF A0 (IF A2 D7 D3) D0).
```

Note that this individual performs better than the best individual from generation 0 for two reasons. First, this individual considers two of the three address bits (A0 and A2) in deciding which data bit to choose as output, whereas the best individual in generation 0 considered only one of the three address bits (A0). Second this best individual from generation 1 incorporates three of the eight data bits as its output, whereas the best individual in generation 0 incorporated only two of the eight potential data bits as output.

Although still far from perfect, the best individual from generation 1 is less blind and more complex than the best individual of the previous generation.

Note that the size and shape of this best scoring individual from generation 1 differs from the size and shape of the best scoring individual from generation 0. The progressive change in size and shape of the individuals in the population is a characteristic of the genetic programming paradigm.

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

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

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

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

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

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

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

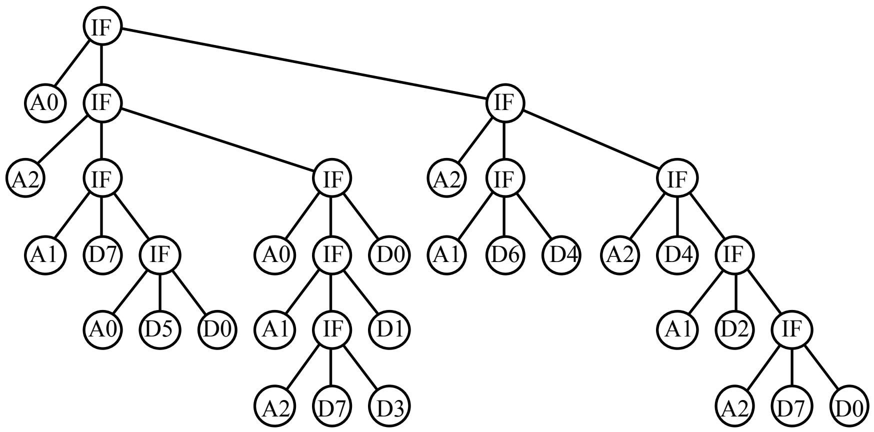
In generation 6, four individuals attain a score of 1792 hits. The high point of the histogram has moved to 1536 hits. In generation 7, 70 individuals score 1792 hits.

In generation 8, there are four best-of-generation individuals. They all attain a score of 1920 hits. The mode (high point) of the histogram has moved to 1664 and 1672 individuals share this value. Moreover, an additional 887 individuals score 1792.

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

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

This 100% correct individual is depicted graphically below:



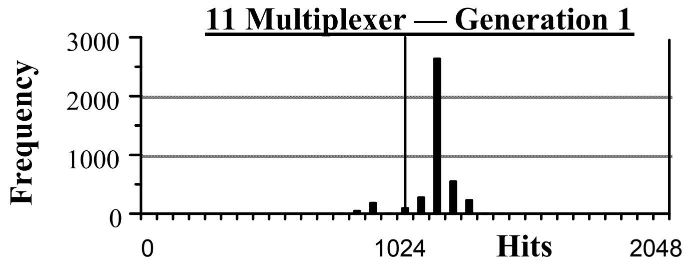
This 100% correct individual is a hierarchical structure consisting of 37 points (i.e. 12 functions and 25 terminals). Note that the size and shape of this solution emerged from the genetic programming paradigm. This particular size and this particular hierarchical structure was not specified in advance. Instead, it evolved as a result of reproduction, crossover, and the relentless pressure of fitness. In generation 0, the best single individual in the population had 12 points. The number of points in the best single individual in the population varied from generation to generation. It was 7, 16, 10, 10, 25, 48, 46, 54, and 60 for generations 1 through 9, respectively.

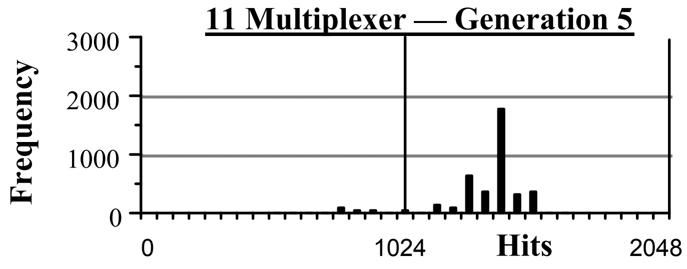
This 100% correct individual can be simplified to

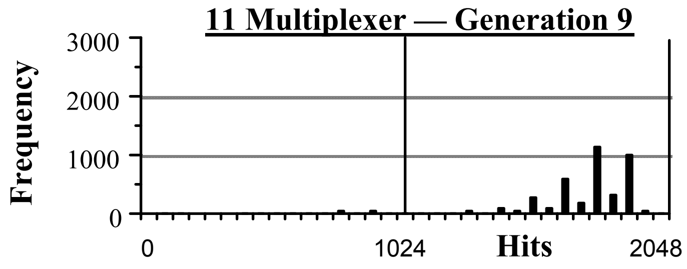
```
(IF A0 (IF A2 (IF A1 D7 D5) (IF A1 D3 D1))
       (IF A2 (IF A1 D6 D4) (IF A1 D2 D0))).
```

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

Table l shows, side by side, the hits histograms for the generations 1, 5, and 9 of this run. As one progresses from generation to generation, note the left-to-right "slinky" undulating movement of the center of mass of the histogram and the high point of the histogram. This movement reflects the improvement of the population as a whole as well as the best single individual in the population. There is a single 100% correct individual with 2048 hits is at generation 9; however, because of the scale of the vertical axis of this histogram, it is not visible in a population of size 4000.







Further insight can be gained by studying the genealogical audit trail of the process. This audit trail consists of a complete record of the details of each operation that is performed. For the operations of fitness proportionate reproduction and crossover, the details consist of the individual(s) chosen for the operation and, for crossover, the particular points chosen within both participating individuals.

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

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

Even with the use of pointers from descendants back to ancestors, construction of a complete genealogical audit trail is exponentially expensive in both computer time and memory space. Note that the audit trail must be constructed for each individual of each generation because the identity of the l00% correct individual(s) eventually solving the problem is not known in advance. Thus, there are 4000 audit trails. By generation 9, each of these 4000 audit trails recursively incorporates information about operations involving up to 1024 ancestors. The audit trail for the single 100% correct individual of interest in generation 9 alone occupies about 27 densely-printed pages.

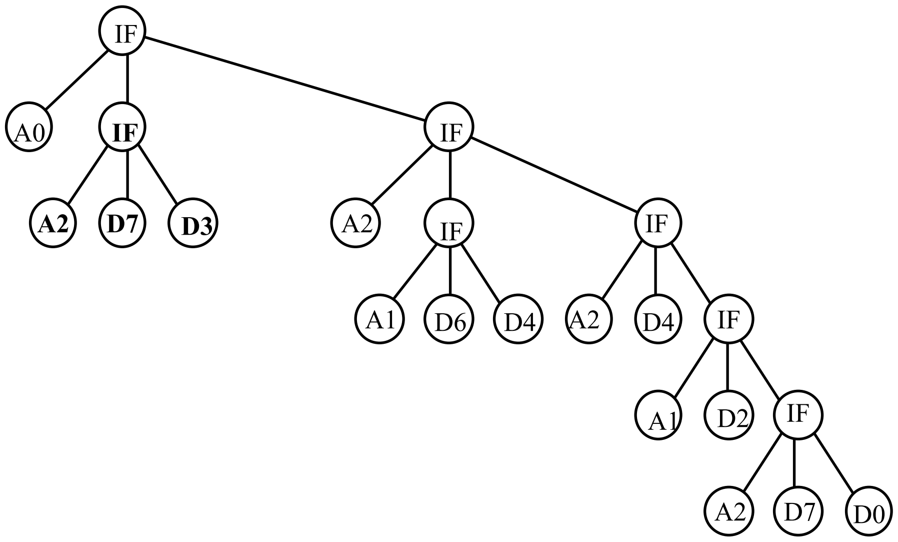
The creative role of crossover and case-splitting is illustrated by an examination of the genealogical audit trail for the l00% correct individual emerging at generation 9. The l00% correct individual emerging at generation 9 is the child resulting from the most common genetic operation used in the process, namely crossover. The first parent from generation 8 had rank location of 58 (out of 4000, with a rank of 0 being the very best) in the population and scored 1792 hits (out of 2048). The second parent from generation 8 had rank location 1 and scored 1920 hits. Note that it is entirely typical that the individuals selected to participate in crossover have relatively high rank locations in the population since crossover is performed among individuals in a mating pool created proportional to fitness.

The first parent from generation 8 (scoring 1792) was

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

Note that this first parent starts by examining address bit A0. If A0 is T, the emboldened and underlined portion then examines address bit A2. It then, partially blindly, makes the output equal D7 or D3 without even considering address bit A1. Moreover, the underlined portion of this individual does not even contain data bits D1 and D5. On the other hand, when A0 is NIL, this first parent is 100% correct. In that event, it examines A2 and, if A2 is T, it then examines A1 and makes the output equal to D6 or D4 according to whether A1 is T or NIL. Moreover, if A2 is NIL, it redundantly retests A2 and then correctly makes the output equal to `(IF A1 D2 D0)`.

This first parent from generation is graphically depicted below:



Note that the 100% correct portion of this first parent, namely, the sub-expression

```
(IF A2 (IF A1 D6 D4) (IF A2 D4 (IF A1 D2 (IF A2 D7 D0))))))
```

is itself a 6-multiplexer which tests A2 and A1 and correctly selects amongst D6, D4, D2, and D0. This becomes clear if we simplify this sub-expression to

```
(IF A2 (IF A1 D6 D4) (IF A1 D2 D0))
```

In other words, this imperfect first parent handles the even-numbered data bits correctly, but only partially correctly handles the odd-numbered data bits. The tree representing this first parent has 22 points. The crossover point chosen at random at the end of generation 8 was point 3 and corresponds to the second occurrence of the function IF. The crossover fragment consists of the emboldened and underlined sub-expression

`(IF A2 D7 D3)`.

The second parent from generation 8 (scoring 1920 hits) was

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

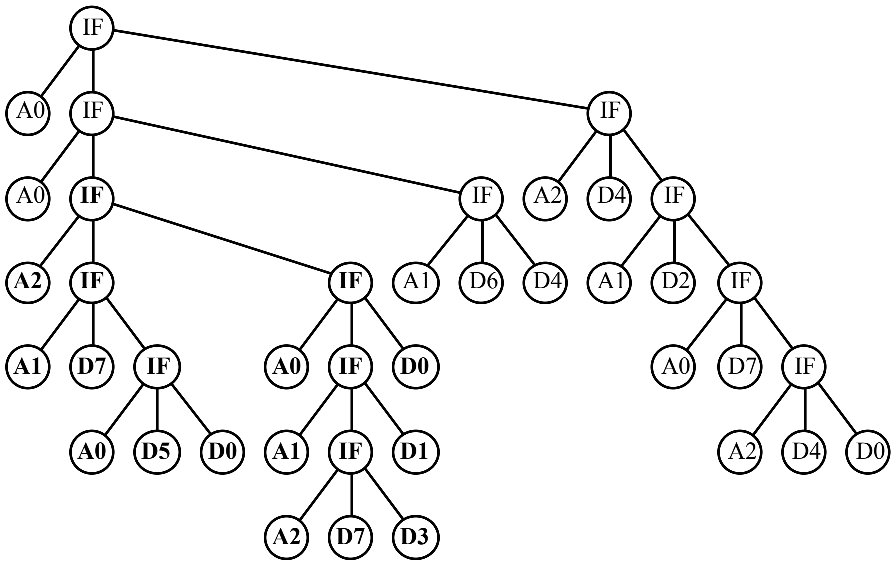
This second parent has 40 points. The crossover point chosen at random for this second parent was point 5. The crossover fragment consists of the emboldened, underlined sub-expression. It correctly handles the case when A0 is T (i.e. the odd numbered addresses). It makes the output equal to D7 when the address bits are 111; it makes the output equal to D5 when the address bits are 101; it makes the output equal to D3 when the address bits are 011; and it makes the output equal to D1 when the address bits are 001.

Note that the 100% correct portion of this second parent, namely, the sub-expression

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

is itself a 6-multiplexer.

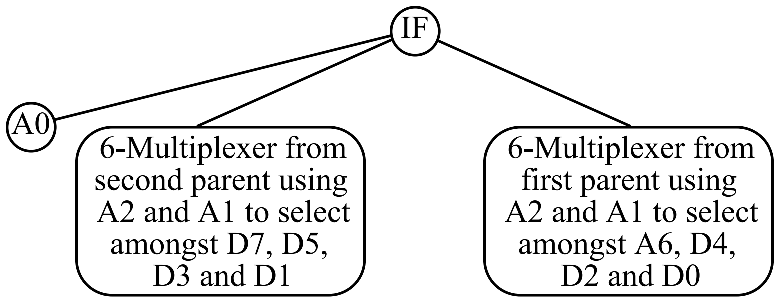
This second parent from generation 8 is graphically depicted below:



This embedded 6-multiplexer tests A2 and A1 and correctly selects amongst D7, D5, D3, and D1 (i.e. the odd numbered data bits). This fact becomes clearer if we simplify this sub-expression of this second parent to the following:

```
(IF A2 (IF A1 D7 D5) (IF A1 D3 D1)
```

This case splitting is graphically demonstrated by the following restatement of the 100% correct offspring into two 6-multiplexers:


IF
A0
6-Multiplexer from second parent using A2 and A1 to select amongst D7, D5, D3 and D1
6-Multiplexer from first parent using A2 and A1 to select amongst A6, D4, D2 and D0

In other words, this imperfect second parent handles part of its environment correctly and part of its environment incorrectly. In particular, it handles the odd-numbered data bits correctly but only partially correctly handles the even-numbered data bits.

Even though neither parent is perfect, these two imperfect parents contain complementary, co-adapted portions which, when mated together, produce a 100% correct offspring individual. In effect, the creative effect of the crossover operation blends the two cases of the implicitly "case-split" environment into a single 100% correct solution.

Of course, not all crossovers between individuals are useful and productive. In fact, a large fraction of the individuals produced by the genetic operations are useless. But the existence of a population of alternative solutions to a problem provides the ingredients with which genetic recombination (crossover) can produce some improved individuals. The relentless pressure of natural selection based on fitness then causes these improved individuals to be preserved and to proliferate. Moreover, genetic variation and the existence of a population of alternative solutions to a problem makes it unlikely that the entire population will become trapped in local maxima.

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

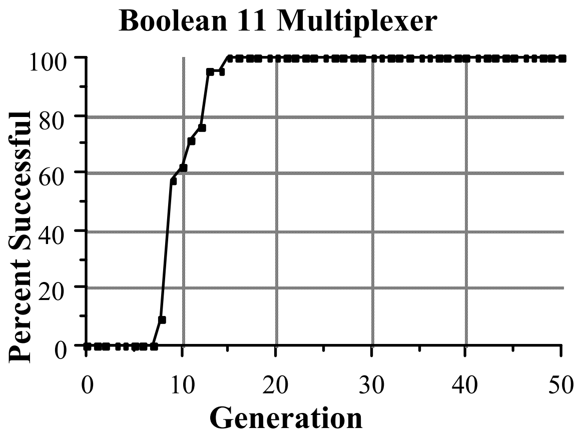
As one traces the ancestry of the 100% correct individual created in generation 9 deeper back into the genealogical audit tree (i.e. towards earlier generations), one encounters parents scoring generally fewer and fewer hits. But if we look at the sequence of hits in the forward direction, we see localized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

The genetic programming paradigm (as with genetic algorithms in general) contains probabilistic steps at several different points. As a result, we rarely obtain a solution to a problem in the precise way we anticipate and we rarely obtain the precise same solution twice. We can measure the number of individuals that need to be processed by a genetic algorithm to produce a desired result (i.e. 2048 matches) with a certain probability, say 99%. Suppose, for example, a particular run of a genetic algorithm produces the desired result with only a probability of success p_s after a specified choice (perhaps arbitrary and non-optimal) of number of generations N_{gen} and population of size N. Suppose also that we are seeking to achieve the desired result with a probability of, say, $z = 1 - \varepsilon = 99\%$. Then, the number K of independent runs required is

$$K = \frac{\log (1-z)}{\log (1-p_s)} = \frac{\log \varepsilon}{\log (1-p_s)}, \text{ where } \varepsilon = 1-z.$$

We ran 21 runs of the Boolean 11-multiplexer problem with a population size of 4,000 and 51 generations. We found that the probability of success p_s was 100% by generation 15 for all runs. Each run was aborted as soon as a perfect solution was found. A total of 980,000 individuals were processed. This is an average of 46,667 individuals per run.

Since each run involves 4,000 individuals, a perfect solution is found in an average of 11.67 generations. The graph below shows the probability of success p_S of a run for various numbers of generations between 0 and 50 for a population size of 4000.



5. Hierarchies and Default Hierarchies

Note that the result of the genetic programming paradigm is always hierarchical. As we saw above, the solution to the 11-multiplexer problem was a hierarchy consisting of two 6-multiplexers. In one run where we applied the genetic programming paradigm to the simpler Boolean 6-multiplexer, we obtained the following 100% correct solution

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

This solution to the 6-multiplexer is also a hierarchy. It is a hierarchy that correctly handles the particular fitness cases where (AND A0 A1) is true and then correctly handles the remaining cases where (AND A0 A1) is false.

Default hierarchies often emerge from the genetic programming paradigm. A default hierarchy incorporates partially correct sub-rules into a perfect overall procedure by allowing the partially correct (default) sub-rules to handle the majority of the environment and by then dealing in a different way with certain specific exceptional cases in the environment. The S-expression above is also a default hierarchy in which the output defaults to

```
(IF A0 D1 (IF Al D2 D0))
```

three quarters of the time. However, in the specific exceptional fitness case where both address bits (A0 and A1) are both T, the output is the data bit D3. Default hierarchies are considered desirable in induction problems (Holland 1986, Wilson 1987a) because they are often parsimonious and they are a human-like way of dealing with situations. Wilson’s noteworthy BOOLE experiments (1987a) originally found a set of eight if-then classifier system rules for the Boolean 6-multiplexer that correctly (but tediously) handled each particular subcase of the problem. Subsequently, Wilson (1988) modified the credit allocation scheme and successfully produced a default hierarchy.

6. Non-Randomness of Results

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

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

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

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

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

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

As a matter of fact, we have evidence suggesting that the solutions are appreciably *sparser* in the space of randomly generated compositions of functions than solutions in the original search space of the problem. Consider, for example, the exclusive-or function. The exclusive-or function is the odd parity function with two Boolean arguments. The odd parity function of k Boolean arguments returns T (True) if the number of arguments equal to T is odd and returns NIL (False) otherwise. Whereas there are 2^{64} Boolean functions with six arguments and 2^{2048} Boolean functions with 11 arguments, there are only $2^{2^2} = 2^4 = 16$ Boolean functions with two Boolean arguments. That is, the exclusive-or function is one of only 16 possible Boolean functions with two Boolean arguments and one output. Thus, in the search space of Boolean functions, the probability of randomly choosing T's and NIL's for the 16 lines of a truth table that realizes this particular Boolean function is only 1 in 16.

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

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

In addition, we performed similar experiments on two Boolean functions with three Boolean arguments and one output, namely, the 3-parity function and the 3-multiplexer function (i. e. the If-Then-Else function). There are only $2^{2^3} = 2^8 = 256$ Boolean functions with three Boolean arguments and one output. The probability of randomly choosing a particular combination of T's and NIL's for the 256 lines of a truth table is 1 in 256. If the probability of randomly generating a composition of functions realizing a particular Boolean function with three arguments equaled 1 in 256, we would expect about 39,063 random compositions per 10,000,000 to realize a particular Boolean function. However, after randomly generating 10,000,000 compositions of the functions AND, OR, and NOT, we found only 730 3-multiplexers and no 3-parity functions. That is, our randomly generated compositions of functions realizing the 3-multiplexer function are about 54 times *sparser* than solutions in the original search space of Boolean functions. The 3-parity function is presumably tens of thousands of times scarcer than one in 256..

These three results concerning the 3-parity function, the 3-multiplexer function, and the 2-parity (exclusive-or) function should not be too surprising since the parity and multiplexer functions have long been identified by researchers as functions that often pose difficulties for paradigms for machine learning, artificial intelligence, neural nets, and classifier systems (Wilson 1987a, Wilson 1988, Quinlan 1988, Barto et. al. 1985). In summary, as to these problems, compositions of functions solving the problem are substantially *less dense* than solutions are in the search space of the original problem.

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

In contrast, the problem described in this paper is neither stated nor solved in terms of objects or operators unique or peculiar to LISP. The solution to the Boolean multiplexer function is expressed in terms of ordinary Boolean functions (such as OR, AND, NOT, and IF). Virtually any programming language can express solutions to this problem. The LISP programming language was chosen for use in the genetic programming paradigm primarily because of the many convenient features of LISP (most importantly, the fact that data and programs have the same form in LISP and that this common form corresponds to the parse tree of a computer program). The LISP programming language was *not* chosen because of the presence in LISP of the list as a primitive data type or because of LISP's particular functions for manipulating lists (e.g. CAR and CDR).

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

7. Conclusions

We described the recently developed genetic programming paradigm and enumerated the five major steps for using it. We cited a variety of different problems which this paradigm has successfully solved. We described, in detail, one particular run in which the genetic programming paradigm learned the Boolean 11-multiplexer function and showed how the size and shape of the ultimate solution progressively evolved using genetic programming paradigm. We presented performance statistics for a number of runs that indicate the rapidity of the search technique and that the genetic programming paradigm performs far better than randomly.

References

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, Nichael Lynn. A representation for the adaptive generation of simple sequential programs. In *Proceedings of an International Conference on Genetic Algorithms and Their Applications*. Hillsdale, NJ: Lawrence Erlbaum Associates l985.

De Jong, Kenneth A. Learning with genetic algorithms: an overview. *Machine Learning,* 3(2), 121-138, 1988.

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.

Goldberg, David E., Korb, B., and Deb, K.. Messy genetic algorithms: Motivation, analysis, and first results. *Complex Systems*. Pages 493-530. 3(5) October 1989.

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

Holland, John H. Escaping brittleness: The possibilities of general-purpose learning algorithms applied to parallel rule-based systems. In Michalski, Ryszard S., Carbonell, Jaime G. and Mitchell, Tom M. *Machine Learning: An Artificial Intelligence Approach, Volume II*. P. 593-623. Los Altos, CA: Morgan Kaufmann l986.

Koza, John R. Hierarchical genetic algorithms operating on populations of computer programs. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence (IJCAI)*. San Mateo, CA Morgan Kaufmann 1989.

Koza, John R. *Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems.* Stanford University Computer Science Department Technical Report STAN-CS-90-1314. June 1990. 1990a.

Koza, John R. Genetically breeding populations of computer programs to solve problems in artificial intelligence. In *Proceedings of the Second International Conference on Tools for AI.* Washington. November 6-9, 1990. 1990b

Koza, John R. Genetic evolution and co-evolution of computer programs. In Farmer, Doyne, Langton, Christopher, Rasmussen, S., and Taylor, C. (editors) *Artificial Life II, SFI Studies in the Sciences of Complexity*. Volume XI. Addison-Wesley, Redwood City CA 1991. 1991a.

Koza, John R. Evolution and co-evolution of computer programs to control independent-acting agents. In Meyer, Jean-Arcady and Wilson, Stewart W. *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior.* Paris. September 24-28, 1990. MIT Press, Cambridge, MA, 1991. 1991b.

Koza, John R. Concept formation and decision tree induction using the genetic programming paradigm. In Schwefel, Hans-Paul and Maenner, Reinhard (editors) *Parallel Problem Solving from Nature.* Springer-Verlag, Berlin, 1991. 1991c.

Koza, John R. A genetic approach to econometric modeling. In Bourgine, Paul and Walliser, Bernard. *Proceedings of the 2nd International Conference on Economics and Artificial Intelligence*. Pergamon Press 1991. 1991d.

Koza, John R. Evolving a computer program to generate random numbers using the genetic programming paradigm. In Belew, Rik and Booker, Lashon (editors) *Proceedings of the Fourth International Conference on Genetic Algorithms*. San Mateo, Ca: Morgan Kaufmann Publishers Inc. 1991. 1991e.

Koza, John R. *Genetic Programming*. Cambridge, MA: MIT Press, 1991 (forthcoming). 1991f.

Koza, John R. and Keane, Martin A. Cart centering and broom balancing by genetically breeding populations of control strategy programs. In *Proceedings of International Joint Conference on Neural Networks*, *Washington, January, 1990*. Volume I. Hillsdale, NJ: Lawrence Erlbaum 1990. 1990a.

Koza, John R. and Keane, Martin A. Genetic breeding of non-linear optimal control strategies for broom balancing. In *Proceedings of the Ninth International Conference on Analysis and Optimization of Systems*. Berlin: Springer-Verlag, 1990. 1990b.

Koza, John R. and Rice, James P. Genetic generation of both the weights and architecture for a neural network. In *Proceedings of International Joint Conference on Neural Networks*, *Seattle, July 1991*. IEEE Press 1991. 1991a

Koza, John R. and Rice, James P. A genetic approach to artificial intelligence. In C. G. Langton *Artificial Life II Video Proceedings*. Addison-Wesley 1991. 1991b.

Lenat, Douglas B. AM: An Artificial Intelligence Approach to Discovery in Mathematics as Heuristic Search. PhD Dissertation. Computer Science Department. Stanford University. 1976.

Lenat, Douglas B. The role of heuristics in learning by discovery: Three case studies. In Michalski, Ryszard S., Carbonell, Jaime G. and Mitchell, Tom M. *Machine Learning: An Artificial Intelligence Approach, Volume I*. P. 243-306. Los Altos, CA: Morgan Kaufman l983.

Lenat, Douglas B. and Brown, John Seely. Why AM and EURISKO appear to work. *Artificial Intelligence*. 23 (1984). 269-294.

Quinlan, J. R. Induction of decision trees. *Machine Learning* 1 (1), 81-106, 1986.

Quinlan, J. R. An empirical comparison of genetic and decision-tree classifiers. *Proceedings of the Fifth International Conference on Machine Learning*. San Mateo, CA: Morgan Kaufmann. 1988.

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

Wilson, Stewart. W. Classifier Systems and the animat problem. *Machine Learning*, 3(2), 199-228, 1987. 1987a

Wilson, S. W. Hierarchical credit allocation in a classifier system. *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, 217-220, 1987. 1987b.

Wilson, Stewart W. Bid competition and specificity reconsidered. *Journal of Complex Systems*. 2(6), 705-723, 1988.