1. .............. INTRODUCTION AND OVERVIEW 1
2. .............. THE PERVASIVENESS OF PROGRAM INDUCTION 4
3. .............. THE CONVENTIONAL GENETIC ALGORITHM 7
4. .............. OVERVIEW OF THE GENETIC PROGRAMMING PARADIGM 10
4.1. ........... CHOICE OF PROGRAMMING LANGUAGE 14
5. .............. DETAILED DESCRIPTION OF THE GENETIC PROGRAMMING PARADIGM 17
5.1. ........... THE STRUCTURES UNDERGOING ADAPTATION 17
5.2. ........... THE INITIAL STRUCTURES 21
5.3. ........... FITNESS 23
5.4. ........... OPERATIONS FOR MODIFYING STRUCTURES 25
5.4.1. ......... REPRODUCTION 25
5.4.2. ......... CROSSOVER (RECOMBINATION) 26
5.5. ........... THE STATE OF THE SYSTEM 29
5.6. ........... RESULT DESIGNATION 29
5.7. ........... TERMINATION 29
5.8. ........... CONTROL PARAMETERS 30
6. .............. LEARNING OF A BOOLEAN FUNCTION 31
6.1. ........... BOOLEAN 11-MULTIPLEXER 32
6.2. ........... HIERARCHIES AND DEFAULT HIERARCHIES 49
6.3. ........... RESULTS OVER A SERIES OF RUNS 50
6.4. ........... NON-RANDOMNESS OF RESULTS 54
7. .............. ARTIFICIAL ANT PROBLEM 58
8. .............. SOLVING A PAIR OF LINEAR EQUATIONS 66
9. .............. RANDOMIZER 70
10. ............ SEQUENCE INDUCTION 79
11. ............ SIMPLE SYMBOLIC REGRESSION 81
12. ............ SYMBOLIC REGRESSION WITH CONSTANT CREATION 89
13. ............ EMPIRICAL DISCOVERY 90
14. ............ SYMBOLIC INTEGRATION AND DIFFERENTIATION 96
15. ............ SOLVING EQUATIONS 98
15.1. ......... DIFFERENTIAL EQUATIONS 98
15.2. ......... INTEGRAL EQUATIONS 104

15.3.......... INVERSE PROBLEMS 104
15.4.......... GENERAL FUNCTIONAL EQUATIONS 104
16............. PLANNING — BLOCK STACKING 104
16.1.......... CORRECTLY STACKING BLOCKS 109
16.2.......... EFFICIENTLY STACKING BLOCKS 112
17............. OPTIMAL CONTROL 114
18............. MINIMAX STRATEGY FOR A GAME 126
19............. EMERGENT BEHAVIOR FOR AN ANT COLONY 132
20............. ADDITIONAL EXAMPLES 140
21............. CONCLUSIONS 140
22............. ACKNOWLEDGEMENTS 140
23............. REFERENCES 141

# THE GENETIC PROGRAMMING PARADIGM: GENETICALLY BREEDING POPULATIONS OF COMPUTER PROGRAMS TO SOLVE PROBLEMS

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

*ABSTRACT: Many seemingly different problems in machine learning, artificial intelligence, and symbolic processing can be viewed as requiring the discovery of a computer program that produces some desired output for particular inputs. When viewed in this way, the process of solving these problems becomes equivalent to searching a space of possible computer programs for a highly fit individual computer program. The recently developed genetic programming paradigm described herein provides a way to search the space of possible computer programs for a highly fit individual computer program to solve (or approximately solve) a surprising variety of different problems from different fields. In the genetic programming paradigm, populations of computer programs are genetically bred using the Darwinian principle of survival of the fittest and using a genetic crossover (sexual recombination) operator appropriate for genetically mating computer programs. This chapter shows how to reformulate seemingly different problems into a common form (i.e. a problem requiring discovery of a computer program) and, then, to show how the genetic programming paradigm can serve as a single, unified approach for solving problems formulated in this common way.*

## 1. INTRODUCTION AND OVERVIEW

A central question in computer science is “How can computers learn to program themselves to solve problems?”

Existing paradigms for machine learning all involve searching a space of specialized structures for a good or best structure to solve a problem. In each paradigm, the structures involved are distinctly different from computer programs.

- Connectionists envision the solution to a given problem as being a set of real-valued weights. The weights are used to amplify or diminish signals passing along the connecting lines of a neural network. One of several neural network paradigms is used to search for a best set of weights. The neural network using this best set of weights is then used to solve the given problem.
- Selectionists envision the solution to a given problem as being a fixed length character string (i.e. chromosome). Each chromosome represents a possible approach to solving the problem. The conventional genetic algorithm is used to search for a good or best chromosome. The best chromosome specifies the approach to be used to solve the given problem.
- Inductionists envision the solution to a given problem as being a decision tree. Each decision tree classifies each instance of a given problem into a class representing a possible solution to the problem. An inductive method, such as ID3, is used to search for a good or best decision tree. The best decision tree is then used to solve the given problem.

Searching for a specialized structure such as a weight vector, chromosome, or decision tree can be an efficient way to solve certain classes of problems. Moreover, such specialized structures often facilitate mathematical analysis that might otherwise not be possible.

However, these specialized structures are often an unnatural and difficult way of viewing the problem and expressing a solution. In many cases, the flexibility that is really wanted and needed is the flexibility provided by computer programs. Computer programs offer the flexibility to

- perform computations on variables of many different types,
- perform alternative computations conditioned on the outcome of intermediate calculations,
- perform iterations and recursions,
- define computed values and sub-programs so that they can be subsequently re-used, and
- arrange groups of operations into hierarchies.

Producing solutions in hierarchical form is especially important because hierarchies are efficient, easy to understand, and lend themselves to scaling up.

The flexibility we want and need also includes flexibility as to the size, shape, and structural complexity of the solution. The user should not be required to specify the size, shape, and structural complexity of the solution in advance. Instead, the size, shape, and structural complexity of the solution should emerge during the problem solving process. In other words, the size, shape, and structural complexity should be part of the answer produced by a problem solving technique — not part of the question.

Once we realize that what we really want and need is the flexibility offered by computer programs, we start to view searches for specialized structures such as weight vectors, chromosomes, or decision trees as flanking actions against the overall problem of getting computers to learn to program themselves. This chapter is a direct frontal assault on the problem of getting computers to learn to program themselves. Our goal here is to find a computer program to solve the given problem. In particular, we search the space of possible computer programs for a computer program that solves the given problem. The recently developed genetic programming paradigm described herein offers a way to genetically breed a computer program to solves (or approximately solve) the given problem.

This chapter does not offer any mathematical proof that the genetic programming paradigm can always be successfully used to solve all problems of every conceivable type. This chapter does, however, provides a large amount of empirical evidence that this new paradigm can be used to solve a surprisingly variety of seemingly different problems from many different fields.

Specifically, this chapter makes two main points.

- POINT NO. 1: A wide variety of seemingly different problems from many different fields can be reformulated as requiring the discovery of a computer program that produces some desired output when presented with particular inputs. That is, these seemingly different problems can be reformulated as problems of program induction.
- POINT NO. 2: The new genetic programming paradigm described herein provides an efficient and effective way to search the space of possible computer programs for a highly fit

individual computer program to solve (or approximately solve) a wide variety of different problems from many different fields.

We deal with Point No. 1 in Section 2 where we show that many seemingly different problems such as automatic programming, optimal control, planning, finding game playing strategies, symbolic regression, and programming emergent behavior can all be viewed as problems of program induction.

Of course, there is no reason to want to view these seemingly different problems as problems of program induction unless there is some good way to do program induction. Accordingly, the remainder of this chapter deals with Point No. 2. In particular, we describe a single, unified approach to solving the problem of program induction, namely, the genetic programming paradigm. We demonstrate that this new paradigm is effective by presenting a wide variety of different examples from a variety of different fields. Existing paradigms for machine learning or artificial intelligence would probably find it impossible to successfully solve all of these problems. Nonetheless, we use a single, unified approach regardless of whether the example involves automatic programming, optimal control, planning, finding game playing strategies, symbolic regression, or programming emergent behavior.

The goal of this chapter is to establish Point No. 2 with empirical evidence. At some point the reader may begin to feel that the examples being presented have merely become "repetitions" of "the same thing." Indeed, they are. When the reader starts viewing genetic solutions to problems of automatic programming, optimal control, planning, finding game playing strategies, symbolic regression, and programming emergent behavior as "the same thing," this chapter will have succeeded in communicating its main point, namely, Point No. 2.

## 2. THE PERVASIVENESS OF PROGRAM INDUCTION

Program induction involves the inductive discovery of a computer program, from the space of possible computer programs, that produces some desired output when presented with some particular input.

As an example of program induction, consider the pair of linear equations

$$a_{11}x_1 + a_{12}x_2 = b_1$$
$$a_{21}x_1 + a_{22}x_2 = b_2$$

in two unknown variables $x_1$ and $x_2$. The well known mathematical formula for solving such a pair of equations starts with six given values (i.e. $a_{11}$, $a_{12}$, $a_{21}$, $a_{22}$, $b_1$, and $b_2$) as its input and produces, as its output, the values of the two unknown variables (i.e. $x_1$ and $x_2$) that satisfy the pair of equations. Program induction involves finding the computer program that implements this well known mathematical formula. The process of program induction uses a finite sampling of combinations of the inputs and correct outputs to induce a computer program. The induced computer program should produce the correct output for any case from the original finite sampling and should also generalize so as to produce the "correct" output for a previously unseen case.

A wide variety of seemingly different problems can be reformulated as a problem of program induction. This fact is obscured by the different terminology used in different fields to describe the concept of program induction.

Depending on the terminology of the particular field involved, the computer program may be called a formula, plan, control strategy, computational procedure, model, decision tree, game-playing strategy, transfer function, mathematical expression, or, perhaps merely, a composition of functions.

Similarly, the inputs to the computer program may be called the sensor values, state variables, independent variables, attributes, information to be processed, input signals, input values, known variables, or, perhaps merely, arguments of a function.

The output from the computer program may be called a dependent variable, a control variable, category, class, move, decision, action, effector, result, output signal, output values, unknown variables, or, perhaps merely, the value returned by a function.

Regardless of the differences in terminology, the problem of discovering a computer program that produces some desired output when presented with particular inputs is common to many seemingly different situations. Several examples follow.

## SYMBOLIC REGRESSION

Symbolic regression (symbolic function identification) involves finding a mathematical expression, in symbolic form, that provides a good, best, or perfect fit between a given finite sampling of values of the independent variables and the associated values of the dependent variables. That is, symbolic regression involves finding a model that fits a given sample of data.

When the variables are real valued, symbolic regression involves finding *both* the functional form *and* the numeric coefficients for the model. Symbolic regression differs from conventional linear, quadratic, or polynomial regression. The latter merely involve finding the numeric coefficients for a function whose form (linear, quadratic, or polynomial) has been pre-specified.

In any case, the mathematical expression being sought in symbolic regression can be viewed as a computer program which takes the values of the independent variables as input and produces the values of the dependent variables as output.

Symbolic regression of real-valued variables is discussed in Section 11. It is later discussed with constant creation in Section 12. If the data is noisy data from the real world, this problem of finding the model from the data is often called empirical discovery (See Section 13). If the independent variable consists of non-negative integers, symbolic regression may be called sequence induction (See Section 10). Machine learning of the Boolean 11-multiplexer function is symbolic regression applied to a function with a Boolean range and domain (Section 6).

**PLANNING**

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

An example of a planning problem involves discovering a robotic action plan to navigate an artificial ant along an irregular trail to find the food lying along the trail (Section 7). Another example involves discovering a plan for stacking blocks in the correct order (Section 16).

The desired plan in a planning problem can be viewed as a computer program. The computer program takes the information from the sensors or detectors as its input and produces effector actions as its output. The effector actions, in turn, cause a change in the state of the objects of the system.

**OPTIMAL CONTROL**

Optimal control involves finding a control strategy that uses the state variables of a system to choose the control variables which will change the state of the system to the desired target state with minimal cost.

An example of an time-optimal control problem involves discovering a bang-bang control strategy for balancing a broom on a moving cart in minimal time. The state variables of the system are the position of the cart, velocity of the cart, angle of the broom, and angular velocity of the broom. The control variable is the bang-bang force that may be applied to the system.

The desired control strategy in an optimal control problem can be viewed as a computer program. The computer program takes the state variables of the system as its input and produces values of the control variable as its output. The control variables, in turn, cause a change in the state of the system (Section 17).

## AUTOMATIC PROGRAMMING

Randomizers (i.e. computer programs that convert a sequence of consecutive integers into a high entropy sequence of random digits) are considered difficult to write. Automatic programming of a randomizer is another example of program induction. The desired computer program takes a sequence of consecutive integers as its input and produces a sequence of bits with high entropy as its output (Section 9).

## MINIMAX STRATEGY FOR PLAYING A GAME

Game playing requires finding a strategy that specifies what move a player is to make at each point in the game, given the known information about the game.

In a game, the known information may be an explicit history of the previous moves by the players or an implicit history of previous moves in the form of a current "state" of the game (e.g. in chess, the position of each piece on the board).

The game-playing strategy can be viewed as a computer program which takes the known information about the game as its input and produces a move as its output (Section 18).

## EMERGENT BEHAVIOR

Emergent behavior involves the repetitive application of seemingly simple rules which lead to complex overall behavior. The evolution of the sets of rules which produce emergent behavior is a problem in program induction.

An example is the problem of finding a set of rules for controlling the behavior of an individual ant which, when simultaneously executed in parallel by all the ants in the colony, causes the ants to work together to locate and transport available food to the nest. The computer program (i.e. set of rules) being sought takes the sensory input of each ant as input and produces actions by the ants as output (Section 19).

## 3. THE CONVENTIONAL GENETIC ALGORITHM

Before discussing the genetic program paradigm, we will first describe the conventional genetic algorithm.

In nature, the evolutionary process occurs when the following four conditions are satisfied:

- An entity must have the ability to reproduce (or approximately reproduce) itself.
- There must be a population of such self-reproducing entities.
- There must be some variety amongst the entities in the population.
- There must be some difference in ability to survive in the environment associated with the variety.

This variety is manifested as variation in the chromosomes of the entities which, in turn, is reflected in variation in the structure and behavior of the entities. Variation in structure and behavior is, in turn, reflected as differences in the rate of survival. The existence of some variability that has some differential effect on the rate of survivability is almost inevitable. Thus, the presence of the first condition (i.e. self-reproducibility) typically is sufficient to start the evolutionary process.

Thus, in nature, entities with the ability to reproduce themselves become better able to perform tasks in their environment because the more fit entities survive at a differentially higher rate. Over a period of time and many generations, the population becomes more fit in performing its tasks in its environment. That is, the population evolves to higher levels of fitness. The structure of individuals in the population changes over time because of the relentless effects of natural selection.

John Holland's pioneering *Adaptation in Natural and Artificial Systems*[1] described how the evolutionary process in nature can be applied to artificial systems. In particular, Holland's genetic algorithm is a highly parallel mathematical algorithm that transforms a population of individual mathematical objects (typically fixed length character strings patterned after chromosome strings) into a new population using

- the operation of Darwinian fitness proportionate reproduction (survival of the fittest),
- the naturally occurring genetic operation of sexual recombination (crossover), and,
- possibly, a small amount of occasional random mutation.

Many problems can be solved using the conventional genetic algorithm operating on fixed length character strings. An overview of genetic algorithms can be found in Goldberg[2]. Recent research work in the field is reported in Davis[3,4], Schaffer[5], Rawlins[6] and Belew and Booker[7].

The use of fixed length character strings has permitted Holland and others to construct a significant body of theory as to why genetic algorithms work. Much of this theoretical analysis depends on the mathematical tractability of the fixed length character strings as compared with mathematical structures that are more complex and comparatively less susceptible to theoretical analysis. Nonetheless, the use of fixed length character strings as the representational scheme leaves many problems unsettled.

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 its world.

As Davis and Steenstrup[8] point out,

> "In all of Holland's work, and in the work of many of his students, chromosomes are bit strings."

For many problems in machine learning and artificial intelligence, the most natural known representation for a solution is a hierarchical computer program of indeterminate size and shape, as opposed to character strings whose size has been determined in advance. It is sometimes difficult, unnatural, and overly restrictive to represent hierarchies of dynamically varying size and shape with fixed length character strings.

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.

Moreover, string-based representation schemes do not have 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.

The predetermination of the size and shape of solutions and the pre-identification of the particular components of solutions has been a bane of machine learning systems from the earliest times as well as in later efforts in automatic programming. The need for more powerful representations in the genetic algorithm field has been recognized for some time (De Jong[9]).

The structure of the individual mathematical objects that are manipulated by a genetic algorithm can, in fact, be more complex than fixed length character strings. Smith[10] departed from Holland's emphasis on fixed-length character strings by introducing variable length strings (including strings whose elements were if-then rules, rather than single characters).

The introduction of the genetic classifier system (Holland[11] Holland et. al.[12]) continued the trend towards increasing the complexity of the structures undergoing adaptation using the genetic algorithm. The genetic classifier system is a highly parallel cognitive architecture in which the genetic algorithm adaptively modifies a population of if-then rules (whose condition and action parts are fixed length binary strings). Examples of applications of genetic classifier systems include Wilson's[13,14] learning of Boolean function and Forrest's[15] work on parallelization of classifier systems.

Wilson[16] extended Holland’s bucket brigade algorithm for credit allocation in genetic classifier systems by introducing hierarchical credit allocation. Wilson’s hierarchical credit allocation encourages 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 (that is, programs and subroutines) that are needed to solve complex problems.

Goldberg et. al.[17] introduced the messy genetic algorithm 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.

As will be seen, the genetic programming paradigm is a further continuation in the trend towards increasing the complexity of the structures undergoing adaptation using the genetic algorithm. The genetic programming paradigm provides a way to genetically breed a population of hierarchical computer programs to solve problems.

In the genetic algorithm field, Cramer[18] used the genetic algorithm operating on fixed length character strings to generate computer programs with a fixed structure (consisting of an operation and two operands in a hypothetical assembly language) and reported on the highly epistatic nature of the problem.

Fujiki and Dickinson[19] implemented analogs of the genetic operations in the conventional genetic algorithm operating on strings to manipulate the individual clauses of a LISP computer program consisting of a single conditional (COND) statement. The individual if-then clauses of the Fujiki and Dickinson's COND statement were parts of a strategy for playing the iterated prisoner's dilemma game.

## 4. OVERVIEW OF THE GENETIC PROGRAMMING PARADIGM

In Section 2, we showed that it was possible to reformulate a wide variety of seemingly different problems from a wide variety of different fields as problems of program induction. In this section, we describe the recently developed genetic programming paradigm which provides a way to do program induction.

In the genetic programming paradigm, populations of computer programs are genetically bred using the Darwinian principle of survival of the fittest and using a genetic crossover (recombination) operator appropriate for genetically mating computer programs. The structures undergoing adaptation in the genetic programming paradigm are hierarchical computer programs of dynamically varying size and shape.

The process of solving many problems can be reformulated as a search for a most fit individual computer program in the space of possible computer programs. When viewed in this way, the process of solving these problems becomes equivalent to searching a space of possible computer programs for a highly fit individual computer program. In particular, the search space is the hyperspace of computer programs composed of functions and terminals appropriate to the problem domain.

This simulated evolutionary process starts with an initial population of randomly generated computer programs composed of functions and terminals appropriate to the problem domain. The functions may be standard arithmetic operations, standard programming operations, standard mathematical functions, logical functions, and domain-specific functions. Depending on the particular problem, the computer program may be Boolean-valued, integer-valued, real-valued, complex-valued, vector-valued, symbolic-valued, or multiple-valued.

Each individual computer program in the population is measured in terms of how well it performs in the particular problem environment. We call this measure "fitness."

Typically the computer program is run over a number of different fitness cases so that fitness is averaged over a variety of representative different situations.

Unless the problem is so small and simple that it can be solved by blind random search, the computer programs in the initial random generation will have exceedingly poor fitness. Nonetheless, some individuals in the population will turn out to be somewhat more fit than others.

Then, the Darwinian principle of reproduction and survival of the fittest and the genetic operation of sexual recombination (crossover) are used to create a new population of offspring individual computer programs from the current population of individual computer programs. In particular, a genetic process of sexual reproduction between two parental computer programs is used to create offspring computer programs. The two participating parental computer programs are selected in proportion to fitness. The resulting offspring computer program are composed of sub-expressions (sub-trees, sub-programs, sub-routines, "building blocks") from their parents.

Then, the new population of offspring (i.e. the new generation) replaces the old population of parents (i.e. the old generation).

Each new individual in the new population of computer programs is then measured for fitness and the process is repeated.

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

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

Typically the single best individual in the population at the time of termination of a run is designated as the result produced by the genetic programming paradigm. This is called “winner takes all”.

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

The dynamic variability of the computer programs that are developed along the way to a solution is also an important feature of the genetic programming paradigm. In each case, it would be difficult and unnatural to try to specify and restrict the size and shape of the eventual solution in advance. Moreover, the advance specification and restriction of the size and shape of the solution to a problem narrows the window by which the system views the world and might well preclude finding the solution to the problem at all.

Another important feature of the genetic programming paradigm is the absence or relatively minor role of preprocessing of inputs and post-processing of outputs. Both the inputs, intermediate results, and outputs are typically expressed directly in terms of the natural functions and arguments from the problem domain. The output of the genetic programming paradigm comes in the form of a computer program which takes the inputs appropriate to the problem and which produces the outputs required by the problem. This makes the results highly comprehensible and intelligible in the terms of the problem domain.

The genetic programming paradigm is a domain independent (“weak”) method. It provides a single, unified approach to the problem of finding a computer program to solve a problem. In this chapter, we show how to reformulate these seemingly different problems into a common form (i.e. a problem of induction of a computer program) and, then, to describe a single, unified approach for solving problems formulated in this common form.

In summary, the genetic programming paradigm genetically breeds computer programs to solve problems by executing the following three steps:

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

(2) Iteratively perform the following sub-steps 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. The operations are applied to computer program(s) in the population chosen with a Darwinian probability based on fitness.

    (i) Copy existing computer programs to the new population.

    (ii) Create new computer programs by genetically recombining randomly chosen parts of two existing programs.

(3) The single best computer program in the population at the time of termination is designated as the result of the genetic programming paradigm. This result may be a solution (or approximate solution) to the problem.

Figure 1.1 is a flowchart showing the steps of the genetic programming paradigm.

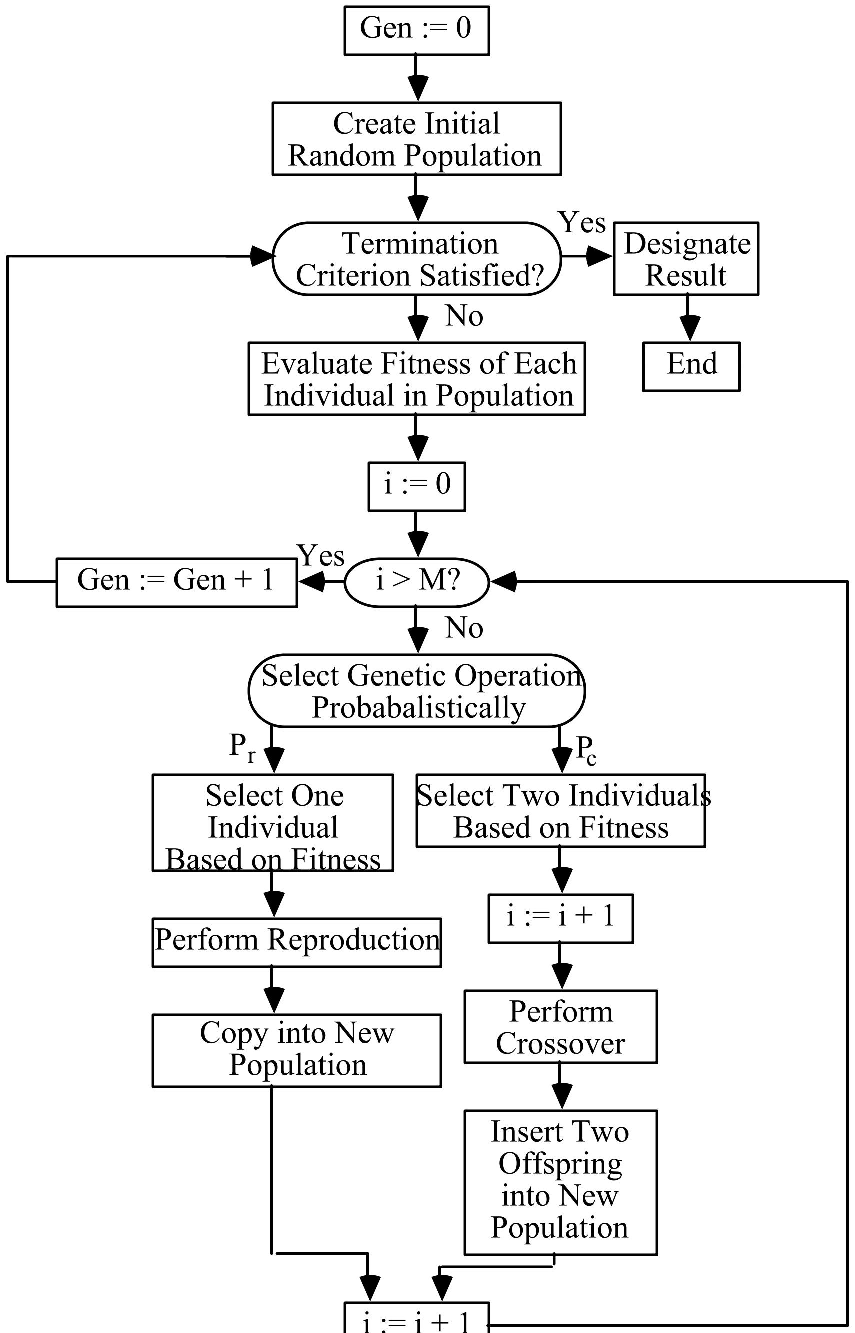

Gen := 0
Create Initial Random Population
Termination Criterion Satisfied?
Yes
Designate Result
End
No
Evaluate Fitness of Each Individual in Population
i := 0
i > M?
Yes
Gen := Gen + 1
No
Select Genetic Operation Probabalistically
$P_r$
$P_c$
Select One Individual Based on Fitness
Perform Reproduction
Copy into New Population
Select Two Individuals Based on Fitness
i := i + 1
Perform Crossover
Insert Two Offspring into New Population
i := i + 1

*Figure 1.1: Flowchart for the genetic programming paradigm*

## 4.1. CHOICE OF PROGRAMMING LANGUAGE

Virtually any programming language (e.g. PASCAL, FORTRAN, C, FORTH, LISP, etc.) is capable of expressing and evaluating the compositions of functions and terminals necessary to implement the genetic programming paradigm. It is possible to implement the genetic programming paradigm using any reentrant programming language that can manipulate computer programs as data and can then compile, link, and execute the new programs or can support an interpreter to execute the new programs created.

We have chosen the LISP (LISt Processing) programming language for the work with the genetic programming paradigm for a number of reasons which we will discuss in detail below. LISP is the most widely known and used example of a functional programming language. All the examples in this chapter will use the Common LISP dialect of LISP herein.

The LISP programming language has only two types of entities, namely atoms and lists. The constant 7 and the variable TIME are examples of atoms in LISP. Lists in LISP consist of an ordered set of items inside a pair of parentheses, such as (+ 1 2) and (FOO BAR).

The LISP compiler and operating system works so as to evaluate whatever it sees. Constant atoms evaluate to themselves when seen by LISP while variable atoms evaluate to their current value. When a list is seen by LISP, the list is evaluated by treating whatever is just inside the opening parenthesis as a function and then causing the application of the function to the remaining items of the list (which are treated as arguments to the function).

We use the name symbolic expression (or, S-expression) for a list or atom in LISP. The S-expressions are the programs of LISP. In fact, S-expressions are the only syntactic form in the LISP programming language.

For example, (+ 1 2) is a LISP S-expression. In this S-expression, the addition function (+) appears just inside the opening parenthesis of the S-expression. This S-expression calls for the application of the addition function + to two arguments (namely, the atoms 1 and 2). The value returned as a result of the evaluation of the S-expression (+ 1 2) is 3.

If any of the arguments are themselves lists (rather than atoms that can be immediately evaluated), Common LISP first evaluates these unevaluated lists (in a recursive, depth-first way, starting from the left) before proceeding.

The LISP S-expression (+ (* 2 3) 4) illustrates the way that computer programs in LISP can be viewed as compositions of functions. This S-expression calls for the application of the addition function (+) to two arguments, namely, the sub-S-expression (* 2 3) and the constant atom 4. In order to complete the evaluation of (+ (* 2 3) 4), LISP must first evaluate (* 2 3). The S-expression (* 2 3) calls for application of the multiplication function (*) to the two constant atoms 2 and 3. The entire S-expression (+ (* 2 3) 4) illustrates the composition of functions.

Now consider LISP symbolic expression (S-expression)

```
(+ 1 2 (IF (> TIME 10) 3 4))
```

This simple S-expression illustrates how functional languages, such as LISP, enable us to view computer programs (with their conditional actions) as compositions of functions and arguments. In the sub-expression (> TIME 10), the function > is applied to the variable atom TIME and the constant atom 10. The sub-expression (> TIME 10) then evaluates to either T (True) or NIL (False) depending on the current value of the variable atom TIME.

The logical value returned by the sub-expression (> TIME 10) becomes the first argument of the function IF. The function IF is a function of three arguments. It returns the result of evaluating its second argument (i.e. the constant atom 3) if its first argument evaluates to T (True, non-NIL) and it returns the result of evaluating its third argument (i.e. the constant atom 4) if its first argument is NIL. Thus, this S-expression evaluates to either 6 or 7 depending on whether the current value of the variable atom TIME is, or is not, greater than 10.

Any LISP S-expression can be graphically depicted as a rooted, point-labeled tree with ordered branches. The tree corresponding to this LISP S-expression is shown in Figure 1.2.

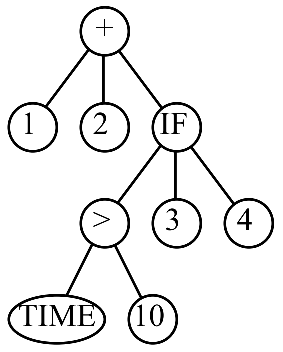


*Figure 1.2 Rooted, point-labeled tree with ordered branches corresponding to the LISP S-expression (+ 1 2 (IF (> TIME 10) 3 4))*

In this graphical depiction, the three internal points of the tree are labeled with functions (e.g. +, IF and >). The six external points (leaves) of the tree are labeled with terminals (e.g. the variable atom TIME and the constant atoms 1, 2, 10, 3, and 4). The root of the tree is labeled with the function (i.e.+) appearing just inside the opening parenthesis of the S-expression.

Note that this tree form of a LISP S-expression is equivalent to the parse tree which many compilers construct internally to represent a given computer program.

The reasons for choosing the LISP programming language for the work with the genetic programming paradigm are as follows:

First, in the LISP programming language, both programs and data have the same form (i.e. the S-expressions). Thus, it is both possible and convenient to treat a computer program in the genetic population as *data* so that it can first be genetically manipulated. Then, it is both possible and convenient to immediately execute the result of the manipulation as a *program*.

Second, the above-mentioned common form for both programs and data in LISP (i.e. the S-expressions) is equivalent to the parse tree for the computer program. In spite of their outwardly different appearance and syntax, most “compiled” programming languages internally convert, at the time of compilation, a given program into a parse tree representing the underlying composition of functions and terminals of that program. In most programming languages, this parse tree is not accessible (or at least not conveniently accessible) to the programmer. And, if it were accessible, it would have a different appearance and syntax than the programming language itself. We need access to the parse tree because we want to genetically manipulate the parts of computer programs (i.e. sub-trees of the parse tree). LISP gives us the ultimate in convenience of access to this parse tree because a LISP program is its own parse tree.

Third, the EVAL function of LISP provides an almost effortless way of executing a computer program that was just created or genetically manipulated.

Fourth, LISP facilitates the programming of structures whose size and shape change dynamically (rather than being predetermined in advance). Moreover, LISP's dynamic storage allocation and garbage collection provide administrative support for programming of dynamically changing structures. The underlying philosophy of all aspects of the LISP programming language is to impose no limitation on programs beyond the limitation inherently imposed by the physical and virtual memory limitations of the computer on which the program is being run. While it is possible to handle structures whose size and shape change dynamically in many programming languages, LISP is especially well suited for this.

Fifth, LISP facilitates the convenient handling of hierarchical structures.

Sixth, software environments offering an unusually rich collection of programmer tools are commercially available for the LISP programming language.

It is important to note that we did *no*t choose the LISP programming language for the work described in this chapter involving the genetic programming paradigm because we intended to make any special use of the “list” data structure from LISP or the list manipulation functions peculiar to the LISP programming language (such as CAR or CDR).

## 5. DETAILED DESCRIPTION OF THE GENETIC PROGRAMMING PARADIGM

Adaptation involves the progressive modification of some structure so that it performs better in its environment. Learning is a form of adaptation for which performance consists of solving a problem. Holland's *Adaptation in Natural and Artificial Systems*[1] provides a general perspective on adaptation and identifies the key features common to every adaptive (or learning) system. In the remainder of this section, we use this perspective to describe the genetic programming paradigm, in detail, in terms of

- the structures that undergo adaptation,
- the initial structures,
- the fitness measure which evaluates the structures
- the operations that are performed to modify the structures,
- the state (memory) of the system at each stage,
- the method for designating a result,
- the method for terminating the process, and
- the parameters that control the process.

## 5.1. THE STRUCTURES UNDERGOING ADAPTATION

In every adaptive system or learning system, some structure or structures are undergoing adaptation.

For non-genetic adaptive algorithms, the structure undergoing adaptation is typically a *single* point in the search space of the problem. For conventional genetic algorithms and the genetic programming paradigm, the structures undergoing adaptation are the individual points, in a *population* of points, from the search space of the problem. That is, the genetic approach involves a parallel search.

The individual structures that undergo adaptation in the genetic programming paradigm are hierarchically structured computer programs. The size, shape, and complexity of these computer programs can dynamically change during the process.

The set of possible structures in the genetic programming paradigm is the set of all possible compositions of functions that can be composed recursively from the available set of $N_{func}$ functions from the function set $F = \{f_1, f_2, ... , f_{N_{func}}\}$ and the available set of $N_{term}$ terminals from the terminal set $T = \{a_1, a_2, ... , a_{N_{term}}\}$. Each particular function f in F takes a specified number z(f) of arguments $b_1, b_2, ..., b_{z(f)}$.

The functions in the function set may be

- arithmetic operations (+, -, *, etc.),
- mathematical functions (such as Sin, Cos, Exp, etc.),
- Boolean operations (such as AND, OR, NOT),
- logical operators (such as If-Then-Else),
- iterative operators (such as Do-Until),
- functions permitting recursion, and

• domain-specific functions.

The terminals are typically either constant atoms (such as the number 3) or variable atoms (representing, perhaps, the inputs, sensors, or state variables of some system).

Consider the function set

```
F = {AND, OR, NOT}
```

and the terminal set

```
T = {D0, D1},
```

where D0 and D1 are Boolean variable atoms that serve as arguments for the functions. We can combine the set of functions and terminals into a combined set C as follows:

```
C  =  F ≈ T  =  {AND, OR, NOT, D0, D1}.
```

Now consider the even parity function with two arguments. This function returns T (True) if an even number (i.e. 0 or 2) of its arguments (i.e. D0 and D1) are T; otherwise, this function returns NIL (False). This Boolean function can be expressed in disjunctive normal form (DNF) and represented by the following LISP S-expression:

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

The rooted, point-labeled tree with ordered branches corresponding to the above S-expression for the even parity function with two arguments is shown in Figure 1.3.

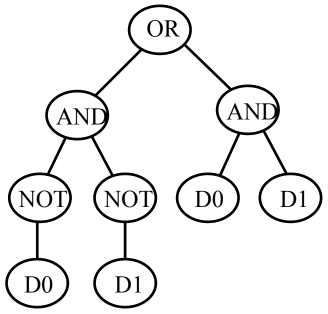


*Figure 1.3 Rooted, point-labeled tree corresponding to the LISP S-expression for the even-parity function (OR (AND (NOT D0) (NOT D1)) (AND D0 D1))*

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

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

The structures that undergo adaptation in the genetic programming paradigm are different than the structures that undergo adaptation in the conventional genetic algorithm operating on strings. The structures that undergo adaptation in the genetic programming paradigm are hierarchical structures. The structures that undergo adaptation in the conventional genetic algorithm are one-dimensional linear strings.

In using the genetic programming paradigm, the terminal set and function set should be selected so as to satisfy two requirements, namely, closure and sufficiency.

**CLOSURE**

As to the closure property, each function in the function set should be well defined for any combination of arguments that may be encountered. These argument values may arise from either a terminal or a function.

For example, if the function set consists of the Boolean functions AND and OR and the terminal set consists only of Boolean variables that can only assume the values of T or NIL, then the closure property will be satisfied. On the other hand, if the arithmetic operation of division is in a function set along with terminals that can assume the numerical value of zero, the closure property will not be satisfied unless some arrangement is made to deal with the situation when division by zero is attempted. One approach is to use the protected division function %. The protected division function % returns one when division by zero is attempted, and, otherwise, returns the normal quotient. Similar arrangements may be required when the square root or logarithm function may be applied to a zero valued variable.

If this closure property does not prevail, we must then address alternatives such as either (1) discarding individuals that do not evaluate to a result that is within the desired domain, or (2) assigning some penalty to the fitness to such individuals and somehow proceeding.

**SUFFICIENCY**

As to sufficiency, needless to say, the set of functions and terminals being used in a particular problem should be selected so as to be capable of solving the problem.

For example, one would not be able to induce Kepler's Third Law for the periods of the planets around the sun if the terminal set contained only the diameter of each planet (as opposed to its distance from the sun) or if the function set contained only addition and subtraction (instead of the functions needed to state the Third Law).

The user of the genetic programming paradigm should know or believe that some composition of the functions and terminals he supplies can yield a solution to the problem. In some domains (e.g. Boolean functions), the requirements are well known. For example, removing the function NOT from the function set F = (AND, OR, NOT} creates an function set that is no longer sufficient for expressing many Boolean functions, including, for example, the even parity function.

The choice of the set of available functions and terminals, of course, directly affects the character of the solutions that can be attained. The available functions and terminals form the basis for generating potential solutions.

For example, if one does symbolic regression on the absolute value function with a function set containing the If-Then-Else function and subtraction, one obtains a solution in the familiar form of a conditional test on x that returns either x or -x depending on whether x is greater or less than zero, respectively. On the other hand, if one does symbolic regression on the absolute value function with a function set containing only the the cosines of odd harmonics and the ordinary arithmetic operations, one instead gets the first few terms of the familiar Fourier series approximation to the absolute value function.

## 5.2. THE INITIAL STRUCTURES

The initial structures in the genetic programming paradigm consist of the individuals in the initial population of individual S-expressions for the problem.

Generation of each individual S-expression in the initial population is done by randomly generating a rooted, point-labeled tree with ordered branches representing the S-expression.

We begin by selecting one of the functions from the set F at random (using a uniform distribution) to be the label for the root of the tree. Note that we restrict selection of the label for the root of the tree to the function set F because we want to generate a hierarchical structure, not a degenerate structure consisting only of a single terminal. For example, in Figure 1.4, the function + (taking two arguments) was selected from a function set F as the label for the root of the tree.

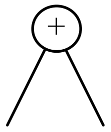


*Figure 1.4 Generation of a random tree might begin with labeling the root of the tree with the + function.*

Whenever a point of the tree is labeled with a function f from F, then z(f) lines, where z(f) is the number of arguments taken by the function f, are created to radiate out from that point. Then, for each such radiating line, an element is randomly selected to be the label for the endpoint of that radiating line.

If a function is chosen to be the label for any such endpoint, the generating process then continues recursively as just described above. For example, in Figure 1.5, the function * from the combined set C = F ≈ T of functions and terminals was selected as the label of the internal non-root point at the end of the first (leftmost) radiating line from the point with the function +. This function * takes two arguments so we show two lines radiating out from the point with the function *.

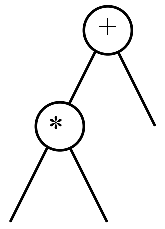


*Figure 1.5 Generation of a random tree might continue with labeling of an internal non-root point with the * function.*

On the other hand, if a terminal is chosen to be the label for any point, that point becomes an endpoint of the tree and the generating process is terminated for that point. For example, in Figure 1.6, the terminal A from the terminal set T was selected to be the label of the first line radiating from the point labeled with the function *.

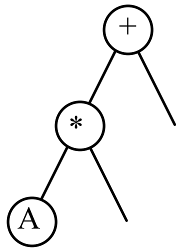


*Figure 1.6 Generation of a random tree might continue with labeling of an external point with the terminal A.*

This generative process can be implemented in several different ways resulting in initial random trees of different sizes and shapes.

The generative method we believe does best over a broad range of problems is a method we call "half ramping." It produces a mixture of trees of various sizes and shapes. This method is used on all problems presented herein. This generative method involves creating an equal number of trees of each depth between two and some maximum depth (which is six for all problems presented herein). The depth of a tree is the length of the longest path from the root to an endpoint.

Then, for each value of depth, 50% of the trees of that depth are created in each of two ways, namely:

- 50% of the trees of the specified depth are "full" in that the length of paths between every endpoint and the root are equal to the maximum. This is accomplished by restricting the random selection of the label for points of the tree at depths smaller than the maximum to the function set F, and restricting the random selection of the label for points at depths equal to the maximum to the terminal set T.

- 50% of the trees of the specified depth are “variably shaped” in that the length of paths between an endpoint and the root is no greater than the specified maximum. This is accomplished by making the random selection of the label for points of the tree at depths smaller than the maximum from the combined set $C = F \approx T$ consisting of the union of the function set F and the terminal set T, while restricting the random selection of the label for points at depths equal to the maximum to the terminal set T.

We use a uniform random probability distribution to make the above selections from the above sets.

Duplicate individuals are unproductive deadwood which waste computational resources and undesirably reduce the genetic diversity of the population. In the genetic programming paradigm, duplicate random individuals are created relatively often when the tree size is small or when the size of the terminal set happens to be large relative to the size of the function set. Thus, after a given individual S-expression is created using the above generative procedure, but before it is actually inserted into the initial population, it is compared to the S-expressions already in the initial population. If the S-expression is a duplicate, it is rejected. The generating process continues until the desired number of unique S-expressions is created.

The variety of a population is the percentage of individuals for which no exact duplicate exists in the population. The variety of the initial random population is 100%.

## 5.3. FITNESS

Each individual in a population is assigned a numerical fitness value as a result of its interaction with its environment. Fitness is the driving force of Darwinian selection in nature. It is, likewise, the driving force for both genetic algorithms and the genetic programming paradigm. In this section we describe four different measures of fitness, namely raw fitness, standardized fitness, adjusted fitness, and normalized fitness.

Raw fitness is the measure of fitness that is stated in the natural terminology of the problem itself. Raw fitness is usually, but not always, evaluated over a set of fitness cases. These fitness cases provide a basis for evaluating the fitness of the S-expressions in the population over different representative situations so that a range of different numerical raw fitness values can be obtained. The fitness cases are typically a small finite sample of the domain space (which is usually very large or infinite). Therefore, the fitness cases must be representative of the domain space as a whole because they form the basis for generalizing the results obtained to the entire domain space.

The definition of raw fitness depends on the problem. For many problems, raw fitness can be defined as the sum of the distances (i.e. errors), taken over all the fitness cases, between the point in the range space returned by the S-expression for the set of arguments for the particular fitness case and the correct point in the range space for the particular fitness case. The S-expression may be, for example, Boolean-valued, integer-valued, real-valued, complex-valued, or symbolic-valued. If the S-expression is integer-valued or real-valued, the sum of distances is the sum of absolute values of the differences (or, if desired, the sum of the squares of the differences) between the numbers involved. When raw fitness is error, the raw fitness r(i,t) of an individual S-expression i in the population of size M at any generational time step t is

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

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

If the S-expression is Boolean-valued or symbolic-valued, the sum of distances is equivalent to the number of mismatches.

For other problems, raw fitness may be something other than error. For example, in optimal control problems, raw fitness may be the cost of an individual control strategy (as measured in time, distance, dollars, etc.). In some problems, raw fitness is a score of some kind (e.g. amount of points scored, benefit achieved, food eaten, subgoals satisfied, etc.).

Note that because raw fitness is stated in the natural terminology of the problem, the better value may either be smaller (as when raw fitness is error) or larger (as when raw fitness is food eaten, benefit achieved, etc.).

The standardized fitness s(i,t) restates the raw fitness so that a lower numerical value is better. If a lower value of raw fitness is better (e.g. when raw fitness represents error), then standardized fitness

$$s(i,t) = r(i,t).$$

If a higher value of raw fitness is better (e.g. when food is being eaten), standardized fitness equals the maximum possible value of raw fitness $r_{max}$ minus the observed raw fitness. That is,

$$s(i,t) = r_{max} - r(i,t).$$

We now define adjusted fitness a(i,t). The adjusted fitness measure a(i,t) is computed from the standardized fitness s(i,t). The adjusted fitness a(i,t) is

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

where s(i,t) is the standardized fitness for individual i at time t.

The adjusted fitness lies between 0 and 1. Unlike standardized fitness, the adjusted fitness is bigger for better individuals in the population.

If no upper bound $r_{max}$ is known (making the above computation of standardized fitness impossible), this step can be omitted and adjusted fitness can be computed directly from raw fitness.

It is not necessary to use adjusted fitness in the genetic programming paradigm. We believe this adjustment is generally helpful and, therefore, we use it consistently on all problems in this chapter. Adjusted fitness has the benefit of exaggerating the importance of small differences in the value of standardized fitness as these values start approaching zero in later generations of a run. Adjusted fitness is especially beneficial if the standardized fitness actually reaches zero when a perfect solution to the problem is found (e.g. as in symbolic regression problems where an error of zero denotes a perfect fit).

We now define normalized fitness n(i,t). The normalized fitness n(i,t) is computed from the adjusted fitness value a(i,t). The normalized fitness n(i,t) is

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

Normalized fitness has three desirable characteristics.

• It ranges between 0 and 1.

• It is larger for better individuals in the population.

• The sum of the normalized fitness values is one.

When we use the phrases proportional to fitness or fitness proportionate in this chapter, we are referring to normalized fitness.

As will be seen, it is also possible for the fitness function to give some weight to secondary factors. Examples of such secondary factors are the efficiency of the S-expression (Section 16.2) and compliance with the initial conditions of a differential equation (Section 15.1).

## 5.4. OPERATIONS FOR MODIFYING STRUCTURES

The two primary operations for modifying the structures undergoing adaptation in the genetic programming paradigm are (1) Darwinian fitness proportionate reproduction and (2) crossover (sexual recombination).

### 5.4.1. REPRODUCTION

The operation of reproduction for the genetic programming paradigm is the basic engine of Darwinian reproduction and survival of the fittest. Each time this operation is performed, it operates on only one parental S-expression and produces only one offspring S-expression. That is, it is an asexual operation.

The operation of reproduction consists of two steps. First, a single S-expression is selected from the population according to some selection rule based on fitness. Second, the individual is copied from the current population into the new population (i.e. the new generation).

There are many different selection rules based on fitness. The most popular selection rule (and the one used herein) is fitness proportionate selection.

When fitness proportionate selection is used as the selection rule in the reproduction operation, if $f(s_i(t))$ is the fitness of individual $s_i$ in the population at generation t, then, each time the reproduction operation is performed, each individual in the population has a probability of being copied into the next generation of

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

When the reproduction operation is performed using fitness proportionate selection as the rule of selection, it is called fitness proportionate reproduction.

Note that the parent remains in the population while selection is performed during the current generation. That is, the selection is done with replacement (i.e. re-selection) allowed. Parents can be selected, *and, in general, are selected*, more than once for reproduction during the current generation. Indeed, the differential rate of survival and reproduction for more fit individuals is an essential part of genetic algorithms.

### 5.4.2. CROSSOVER (RECOMBINATION)

The crossover (sexual recombination) operation for the genetic programming paradigm creates variation in the population by producing new offspring that consist of parts taken from each parent. The crossover operation starts with two parental S-expressions and produces two offspring S-expressions. That is, it is a sexual operation.

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

The operation begins by independently selecting, using a uniform probability distribution, one random point in each parent to be the crossover point for that parent. Note that the number of points in the two parents typically are not equal because the S-expressions in the population are of various shapes and sizes.

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

The first offspring S-expression is produced by deleting the crossover fragment of the first parent from the first parent and then inserting the crossover fragment of the second parent at the crossover point of the first parent. The second offspring is produced in a symmetric manner.

As will be seen, the crossover operation is well-defined and syntactically legal for any two S-expressions and any two crossover points.

For example, consider the two parental LISP S-expressions shown in Figure 1.7. The functions appearing in these two S-expressions are the Boolean AND, OR, and NOT functions. The terminals appearing are the Boolean arguments D0 and D1. Each point of the two S-expressions in this figure has been numbered in a depth-first, left-to-right way.

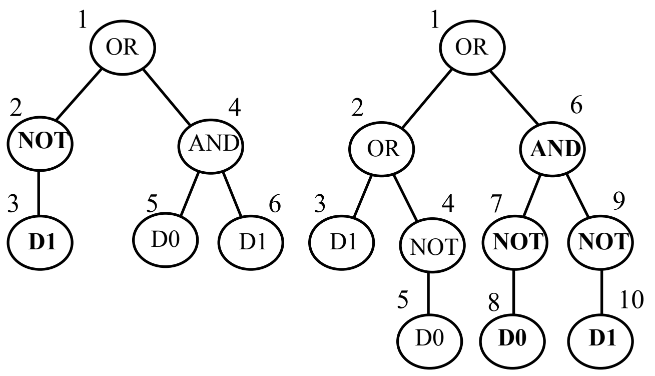


*Figure 1.7 Two parental LISP S-expressions*

Equivalently, in terms of LISP S-expressions, the two parents are

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

and

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

Assume that the points of both trees above are numbered in a depth-first way starting at the left. Suppose that the second point (out of the six points of the first parent) is randomly selected as the crossover point for the first parent. The crossover point of the first parent is therefore the NOT function. Suppose also that the sixth point (out of the 10 points of the second parent) is selected as the crossover point of the second parent. The crossover point of the second parent is therefore the AND function. The underlined and emboldened portions of the two parental S-expressions above are the crossover fragments. Figure 1.8 shows these two crossover fragments.

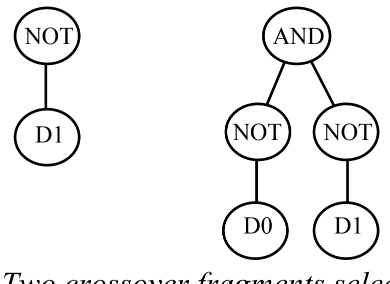


*Figure 1.8 Two crossover fragments selected from the parents from Figure 1.7*

Figure 1.9 shows the two offspring resulting from crossover.

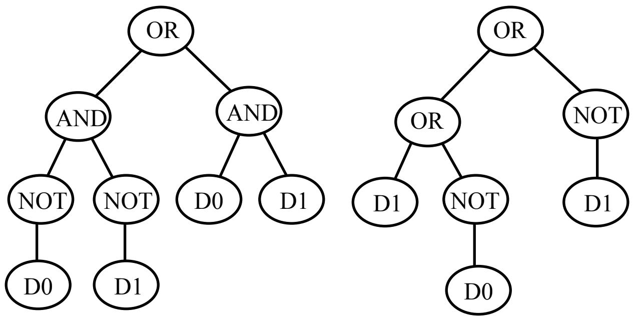


*Figure 1.9 Two offspring produced by the crossover operation using the parents from Figure 1.7 and the crossover fragments from Figure 1.8.*

Note that the first offspring above happens to be a perfect solution for the even parity function, namely

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

The second offspring is

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

Note that because entire sub-trees are swapped and because of the closure property of the functions themselves, this genetic crossover (recombination) operation produces syntactically legal LISP S-expressions as offspring in all situations.

If the root of one parental S-expression happens to be selected as the crossover point, the crossover operation will insert the entire first parent into the second parent at the crossover point of the second parent. That is, in this event, the entire first parent will become a sub-tree within the second parent. In addition, in this event, the crossover fragment of the second parent will then become the second offspring.

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

Note that if an individual mates with itself or two identical individuals mate, the two resulting offspring will generally be different (because the crossover points selected are, in general, different for the two parents). This is in contrast with conventional genetic algorithms that operate on fixed length character strings where the one selected crossover point applies to both parents.

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

If terminals are located at both crossover points selected, the crossover operation merely swaps these terminals from tree to tree. The effect of crossover, in this event, is akin to a point mutation. Thus, occasional point mutation is an inherent part of the crossover operation.

A maximum permissible size (measured via the depth of the tree) is established for offspring created by the crossover operation. This limit prevents large amounts of computer time being expended on a few extremely large individual S-expressions. Of course, if we could execute all the individual S-expressions in the population in parallel (as nature does) in a manner such that the infeasibility of one individual in the population does not disproportionately jeopardize the resources needed by the population as a whole, we would not need such a size limitation. If a crossover between two parents would create an offspring of impermissible size, the contemplated crossover operation is aborted for that offspring and the first of its parents is arbitrarily chosen to be reproduced into the new population.

## 5.5. THE STATE OF THE SYSTEM

The state of the genetic programming paradigm at any point during the process consists *only* of the current population of individuals in the population. There is no additional memory or centralized bookkeeping necessary.

## 5.6. RESULT DESIGNATION

The single best individual in the population at the time of termination of the genetic programming paradigm is typically designated as the result produced by the genetic programming paradigm. This method of results designation is sometimes called "winner takes all" and is used herein.

Note that the very best individual is not guaranteed to be present in the population at the time of termination unless some specific effort is made to preserve this individual (the so-called “elitist” strategy).

Alternately, the entire population at the time of termination of the genetic programming paradigm can be designated as the result produced by the genetic programming paradigm.

## 5.7. TERMINATION

As for termination, as Yogi Berra once said, “It ain't over until it's over, and even then, it's not over.” The genetic programming paradigm parallels nature in that it is a continuing process. As a practical matter, the genetic programming paradigm terminates when either a pre-specified maximum number $N_{gen}$ of generations have been run or when some termination criterion is satisfied.

One possible termination criterion is that the standardized fitness of some individual in the population either equals zero or is within a pre-established neighborhood of zero.

## 5.8. CONTROL PARAMETERS

The genetic programming paradigm is controlled by various parameters, including two major parameters and five minor parameters.

The two major parameters that are used to control the process are the population size M and the number of generations $N_{gen}$ to be run.

First, unless otherwise indicated, the population size was 500 for all problems in this chapter.

Second, unless otherwise indicated, the number of generations was 51 (i.e. an initial random generation plus 50 subsequent generations) for all problems in this chapter. Note, if termination is under control of a problem specific termination criterion, this parameter merely provides an overall maximum number of generations.

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

The values of the five minor parameters are the same for all problems herein.

First, crossover was performed on 90% of the population for each generation. That is, if the population size is 500, then 450 individuals

(225 pairs) from each generation were selected with a probability equal to their normalized fitness (with reselection allowed) to participate in crossover.

Second, fitness proportionate reproduction was performed on 10% of the population on each generation. That is, if the population size is 500, 50 individuals from each generation were selected with a probability equal to their normalized fitness (with reselection allowed).

Third, in selecting crossover points, we used a probability distribution that allocates 90% of the crossover points equally amongst the internal (function) points of each tree and allocates the remainder (i.e. 10%) equally amongst the external (terminal) points of each tree. We believe this distribution promotes the recombining of larger structures. In contrast, a uniform distribution over all points might do an inordinate amount of mere swapping of terminals from tree to tree in a manner more akin to point mutation than the desired recombining of "building block" substructures.

Fourth, a maximum depth of 17 was established for S-expressions created by the crossover operation.

Fifth, a maximum depth of 6 was established for the random individuals generated for the initial population.

## 6. LEARNING OF A BOOLEAN FUNCTION

In the previous sections, we have discussed the background and details of the genetic programming paradigm.

This section, and the remaining sections of this chapter, illustrate the use of the genetic programming paradigm. In each such section, we show how we approached each problem so that it could be solved using the genetic programming paradigm.

The examples have been selected to illustrate a variety of different types of problems from various different areas. The sample problems selected involve functions that are integer-valued, real-valued, Boolean-valued, and symbolic-valued. Some of the problems require iteration for their solution. Some of the problems involve functions whose real functionality is the side effects they cause on the state of the system involved, rather than the actual value returned by the function.

Many of the problems described are benchmark problems that have been the subject of previous study in machine learning, artificial intelligence, induction, neural nets, decision trees, and classifier systems.

Since the genetic programming paradigm is probabilistic, we almost never get the precisely same result twice. Moreover, we almost never get the solution to the problem in the form we contemplated (although these solutions may be equivalent to what we contemplated). For each illustrative problem, we first show the result from one particular run. The showing of one particular run serves to illustrate the representation scheme and the general appearance of results one gets from the genetic programming paradigm. No one particular run and no particular result is truly typical or representative of all runs. In choosing the particular result for each problem, we have avoided showing the "prettiest" result and we have similarly avoided showing the most convoluted result.

We show the amount of computer processing required to produce a solution with 99% probability over a series of runs for selected problems herein. All of the problems presented herein have been repeatedly solved on dozens or hundreds of occasions.

For each problem herein, the author believes that sufficient information is provided herein (or via references) to allow the experiment to be independently replicated to produce substantially similar results (within the limits inherent in any process involving stochastic operations).

We present the first problem below in especially great detail.

## 6.1. BOOLEAN 11-MULTIPLEXER

The problem of machine learning of a Boolean function requires developing a composition of functions that can return the correct value of the function after seeing examples of particular combinations of arguments associated with the correct value of the function. This problem is a special case of the general problem of symbolic function identification (symbolic regression) that will be discussed later in connection with real valued functions.

Boolean functions provide an especially useful test bed for machine learning for several reasons.

First, it is intuitively easy to see how the *structural* components of the S-expression for a Boolean function contribute to the overall *performance* of the Boolean expression. This direct connection between *structure* and *performance* is much harder to comprehend for many of the other problems.

Second, there are fewer practical computer implementation obstacles for Boolean functions than for other problems. For example, with Boolean functions, there is no need to be concerned with error conditions (such as floating point overflows and underflows) arising from randomly generated computer programs and genetically recombined computer programs.

Third, Boolean problems have an easily quantifiable search space.

Let us first consider the problem of learning the Boolean 11-multiplexer function.

The solution of this problem using the genetic programming paradigm 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 and able to correctly handle certain sub-cases of the problem (case-splitting),
- the creative role of crossover in recombining valuable parts of more fit parents, and
- How the nurturing of a large population of 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 importance of not determining in advance the size and shape of ultimate solution or the intermediate results that may contribute to the solution.

The input to the Boolean N-multiplexer function consists of k address bits $a_i$ and $2^k$ data bits $d_i$, where $N = k + 2^k$. That is, the input to the Boolean multiplexer function consists of the $k+2^k$ bits

$$a_{k-1}, \dots , a_1, a_0, \ d_{2^k-1}, \dots , d_1, \ d_0.$$

The value of the Boolean multiplexer function is the Boolean 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 Boolean 11-multiplexer (where $k = 3$), if the three address bits $a_2a_1a_0$ are 110, the multiplexer singles out data bit number 6 (i.e. $d_6$) to be the output of the multiplexer. Figure 1.10 shows a Boolean 11-multiplexer with an input of 1100**1**000000 and the corresponding output of **1**.

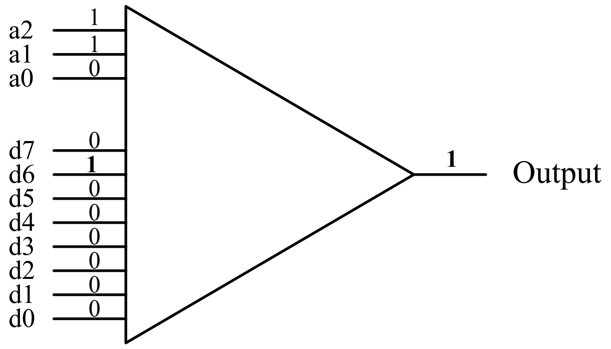


*Figure 1.10 Boolean 11-multiplexer*

There are five major steps involved in using the genetic programming paradigm. These are outlined below.

(1) the set of terminals,
(2) the set of functions,
(3) the fitness function,
(4) the parameters for running the algorithm, and
(5) the criterion for designating a result and terminating a run.

The first major step in setting up the genetic programming paradigm is to select the set of terminals that will be available for constructing the computer programs (S-expressions) that will try to solve the problem. This choice is especially straight-forward for this problem. The terminal set for this problem has 11 elements which correspond to the 11 inputs to the Boolean 11-multiplexer. That is, the terminal set is

```
    T = {A0, A1, A2, D0, D1, ... , D7}.
```

None of these eleven terminals are distinguished as being either address lines or data lines.

The second major step in setting up the genetic programming paradigm is to select the set of functions that will be available for constructing the computer programs (S-expressions) that will try to solve the problem. The set of available functions for this problem is

```
    F = {AND, OR, NOT, IF}
```

having 2, 2, 1, and 3 arguments, respectively. The IF function is the Common LISP function that performs the IF-THEN-ELSE operation.

That is, the IF function returns the results of evaluating its third argument (the “else” clause) if its first argument is NIL (False) and otherwise returns the results of evaluating its second argument (the “then” clause).

Note that this step (performed by the user) of determining the set of primitive functions in the genetic programming paradigm is equivalent to a similar required step in other machine learning paradigms. For example:

- This same determination of primitive functions occurs in the induction of decision trees using ID3 (and its variants) when the user selects the set of attribute-testing functions that can appear at the internal points of the decision tree.
- 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.
- This same user determination occurs in conventional genetic algorithms operating on strings when the user determines how certain external function are to be activated by a chromosome in the user's chosen representation scheme.
- This same user determination occurs in genetic classifier systems when the user selects the external functions that are to be activated by the output interface of the classifier system.

This omnipresent user determination occurs in other machine learning paradigms under various different guises, but is often not explicitly identified as a necessary step by researchers using other paradigms because the researcher often considers the choice of functions to be inherent in the statement of the problem (a view which is especially understandable if the researcher is focusing on only one specific problem in one specific area).

The above function set F of basic logical functions satisfies the closure property. Moreover, this set is known to be sufficient to realize any Boolean function. For this problem and most of the problems herein, the function set is not only minimally sufficient to solve the problem at hand, but contains additional functions.

The search space for this problem is the set of all LISP S-expressions that can be recursively composed of functions from the function set and terminals from the terminal set. Another way to look at the search space is that the Boolean multiplexer function with $k+2^k$ arguments is a particular one of $2^{k+2^k}$ possible Boolean functions of $k+2^k$ arguments. For example, when k=3, then $k+2^k = 11$ and this search space is of size $2^{2^{11}}$. That is, the search space is of size $2^{2048}$, which is approximately $10^{616}$. Note that there are about $10^{68}$ particles in the universe. Every possible Boolean function of $k+2^k$ arguments can be realized by at least one LISP S-expression composed from the functions and terminals above (for example, disjunctive normal form).

The third major step in setting up the genetic programming paradigm is to identify the fitness function for the problem. Fitness is often evaluated over a number of fitness cases. The set of fitness cases must be representative of the problem as a whole. The reader may find it helpful to think of these fitness cases as the "environment" in which the genetic population of computer programs must adapt. There are $2^{11} = 2048$ possible combinations of the 11 arguments $a_0a_1a_2d_0d_1d_2d_3d_4d_5d_6d_7$ along with the associated correct value of the 11-multiplexer function. For this particular problem, we use the entire set of 2048 combinations of arguments as the fitness cases for evaluating fitness. That is, we do not use sampling.

We begin by defining raw fitness in the simplest way that comes to mind using the natural terminology of the problem. The raw fitness of a LISP S-expression in this problem is simply the number of fitness cases (taken over all 2048 fitness cases) where the Boolean value returned by the S-expression for a given combination of arguments is the correct Boolean value. Thus, the raw fitness of an S-expression can range over 2049 different values between 0 and 2048. A raw fitness of 2048 denotes a 100% correct individual S-expression. We define the auxiliary hits measure for this problem to be equal to the raw fitness.

After defining raw fitness for the problem, we proceed to define standardized fitness. Since a bigger value of raw fitness is better, standardized fitness is different from raw fitness for this problem. In particular, standardized fitness equals the maximum possible value of raw fitness $r_{max}$ (i.e. 2048) minus the observed raw fitness. The standardized fitness can also be viewed as 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. The Hamming distance is zero if the Boolean value returned by the S-expression agrees with the correct Boolean value and is one if it disagrees. Thus, the sum of the Hamming distances is equivalent to the number of mismatches.

The fourth major step in using the genetic programming paradigm is selecting the values of certain parameters. A population of size 4000 was chosen for this problem.

Finally, the fifth major step in using the genetic programming paradigm is the criterion for designating a result and terminating a run. In this problem we have a way to recognize a solution when we find it. When the raw fitness is 2048 (i.e. the standardized fitness is zero), we have a 100% correct solution to this problem. Thus, we terminate a run after a specified maximum number of generations $N_{gen}$ (e.g. 51) or earlier if we find an individual with a raw fitness of 2048. For all the problems in this chapter, we will terminate a given run either after 51 generations and we designate the best single individual in the population at the time of termination as the result of the genetic programming paradigm. This is called "winner takes all."

We now illustrate the genetic programming paradigm by discussing one particular run of the Boolean 11-multiplexer in detail.

The process begins with the generation of the initial random population (i.e. generation 0).

Predictably, the initial random population includes a variety of highly unfit individuals. 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 an input through as the output, such as `(NOT (NOT A1))`. Other individuals are inefficient, such as `(OR D7 D7)`. Some of these initial random individuals base their decision on precisely the wrong arguments, such as `(IF D0 A0 A2)`. This individual uses the data bit D0 to decide what output to take. Many of the initial random individuals are partially blind in that they do not incorporate all 11 arguments that are known to be necessary to solve the problem. Some 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. In the valley of the blind, the one-eyed man is king. 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).

The worst individual in the population for the initial random generation was

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

This individual had a standardized fitness of 1280 (i.e. raw fitness of only 768).

As it happens, a total of 23 individuals out of the 4000 in this initial random population tied with the highest score of 1280 matches (i.e. hits) on generation 0. One of these 23 high scoring individuals was the S-expression

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

This individual scores 1280 matches by scoring 512 matches for the one quarter (i.e. 512) of the 2048 fitness cases for which A2 and A1 are both NIL and by scoring an additional 768 matches on 50% of the remaining three quarters (i.e. 1536) of the fitness cases.

This individual has obvious shortcomings. Notably, it is partially blind in that it uses only 3 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, this individual uses one of the three address bits (A0) as the basis for its action. It could easily have done this wrong 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 done this wrong and 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.

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 auxiliary hits measure. 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, for this problem) 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). This well-populated range includes the mode of the distribution which occurs at 1152 matches (hits). There are 1490 individuals with 1152 matches (hits). Figure 1.11 shows the hits histogram of the population for generation 0 of this run of this problem.

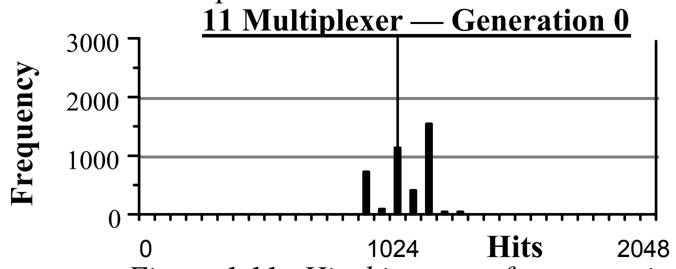


*Figure 1.11 Hits histogram for generation 0*

The Darwinian reproduction operation and the genetic crossover operation are then applied to parents selected from the current population with probabilities proportionate to fitness to breed a new population. 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 because there are 4000 individual points involved.

Although the vast majority of the new offspring are again highly unfit, some of them tend to be somewhat more fit than others. Moreover, over a period of time and many generations, some of them tend to be slightly more fit than those existing in the earlier generation. 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. We typically see this kind of generally improving trend in average standardized fitness from generation to generation. As it happens, in this particular run of this particular problem, 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.

Figure 1.12 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.

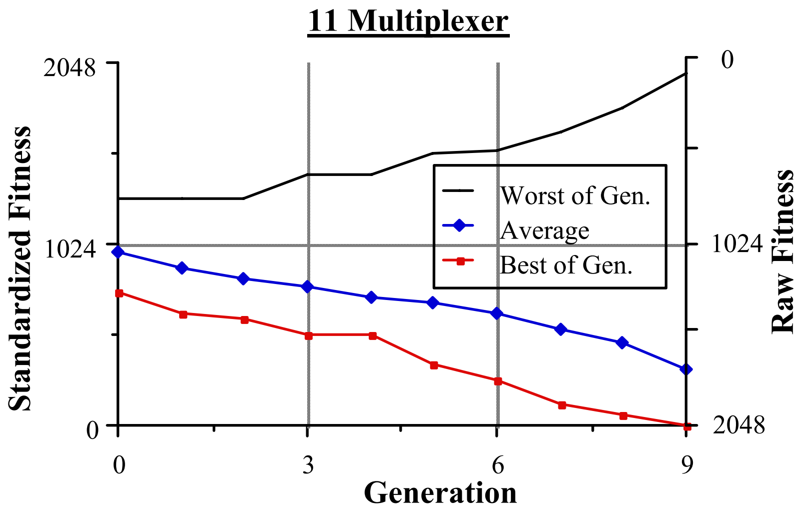


*Figure 1.12 Standardized fitness of worst-of-generation individual, average standardized fitness of population, and standardized fitness of best-of-generation individual for generation 0 through 9.*

In generation 1, the raw fitness for the best single individual in the population rises to 1408 matches (i.e. standardized fitness of 640). Only one individual in the population attained this high score of 1408 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. This best-of-generation individual consists of 7 points, whereas the best-of-generation individual from generation 0 consisted of only 4 points.

In generation 2, the best raw fitness remained at 1408; however, the number of individuals in the population sharing this high score rose from 1 to 21. The high point of the hits histogram advanced from 1152 for generation 0 to 1280 for generation 2. There are 1620 individuals with 1280 hits.

In generation 3, one individual in the population attained a new high score of 1472 matches (i.e. standardized fitness of 576). This individual has 16 points and 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 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 attain this score of 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. 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)))))
```

Figure 1.13 shows the 100% correct individual from generation 9.

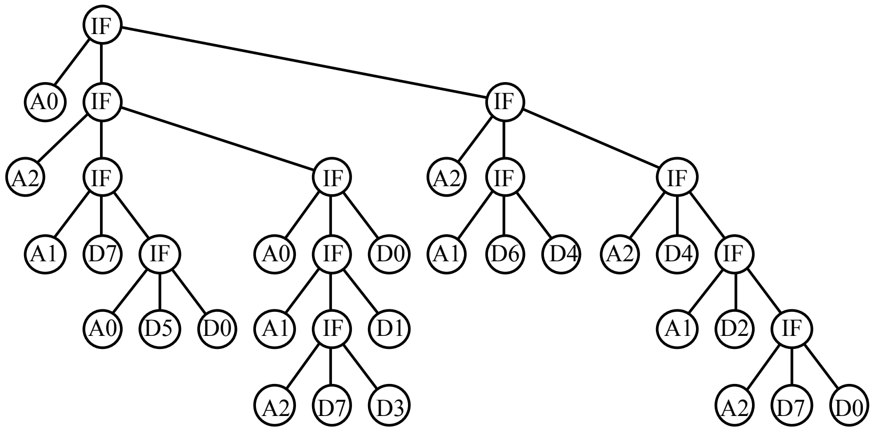


*Figure 1.13 100% correct individual from generation 9*

This 100% correct individual from generation 9 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 4 in generation 0, while it was 37 for generation 9.

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.

Figure 1.14 shows, side by side, the hits histograms for generations 3, 5, 7, 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 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.

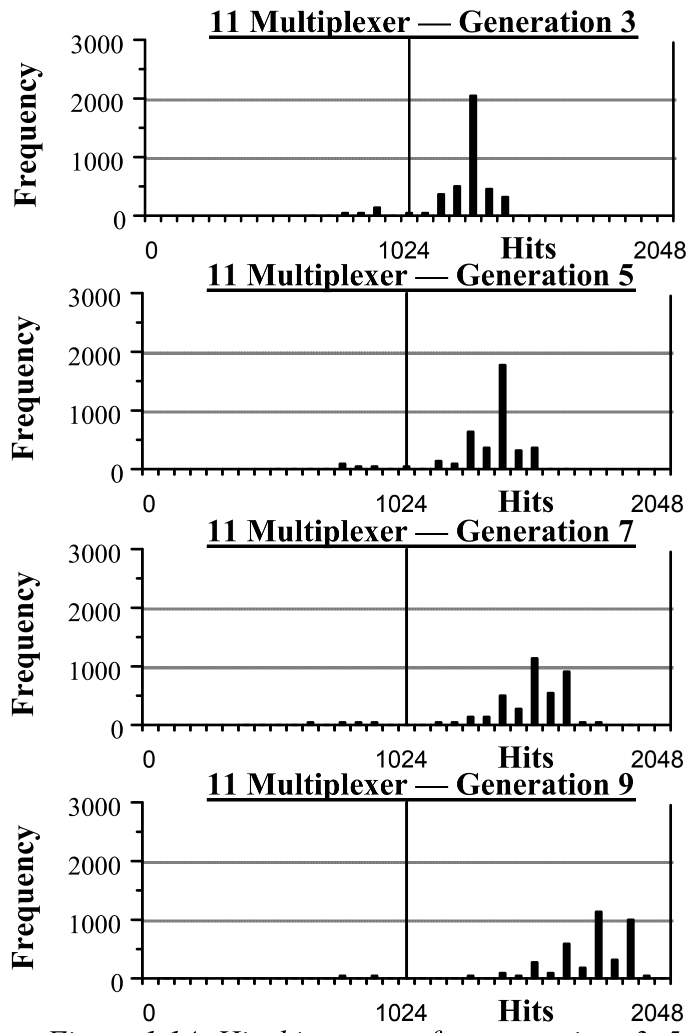


*Figure 1.14: Hits histograms for generations 3, 5, 7, and 9 for the Boolean 11-multiplexer problem*

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 genetic 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 (found by 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 (e.g. crossover point selected), and, finally, a pointer to the audit trail previously assembled 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 and to the extent that an individual crosses over with itself. 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 100% 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)))))).
```

Figure 1.15 shows this first parent from generation 8 .

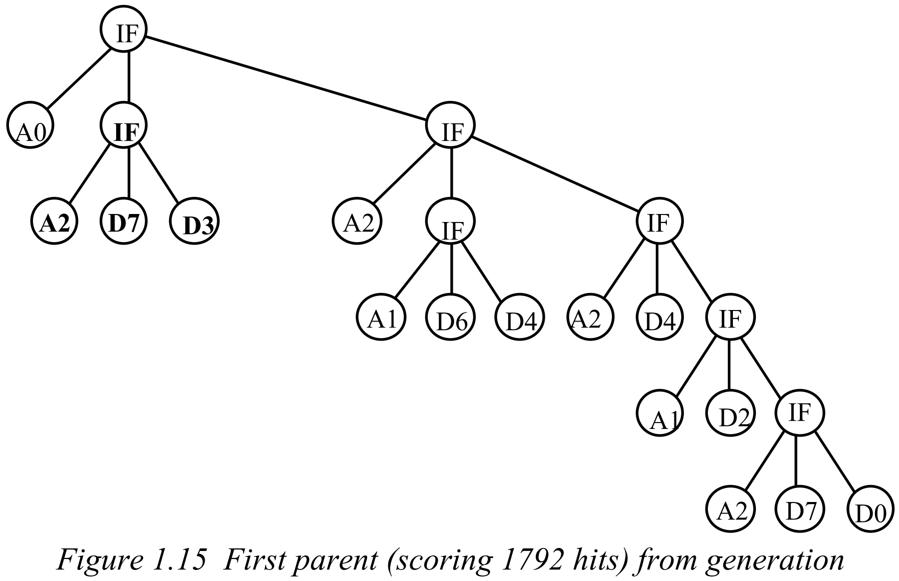


*Figure 1.15 First parent (scoring 1792 hits) from generation 8 for 100% correct individual in generation 9*

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 emboldened and 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 twice retests A2 (unnecessarily, but harmlessly) and then correctly makes the output equal to `(IF A1 D2 D0)`. 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.

This embedded 6-multiplexer tests A2 and A1 and correctly selects amongst D6, D4, D2, and D0. This fact becomes clearer if we simplify this sub-expression by removing the two extraneous tests and removing the D7 (which is unreachable). This sub-expression simplifies to the following:

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

In other words, this imperfect first parent handles part of its environment correctly and part of its environment incorrectly. In particular, this first parent handles the even-numbered data bits correctly. This first parent is partially correct in handling 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. That is, the crossover fragment consists of the incorrect, 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))))))
```

Figure 1.16 shows the second parent from generation 8.

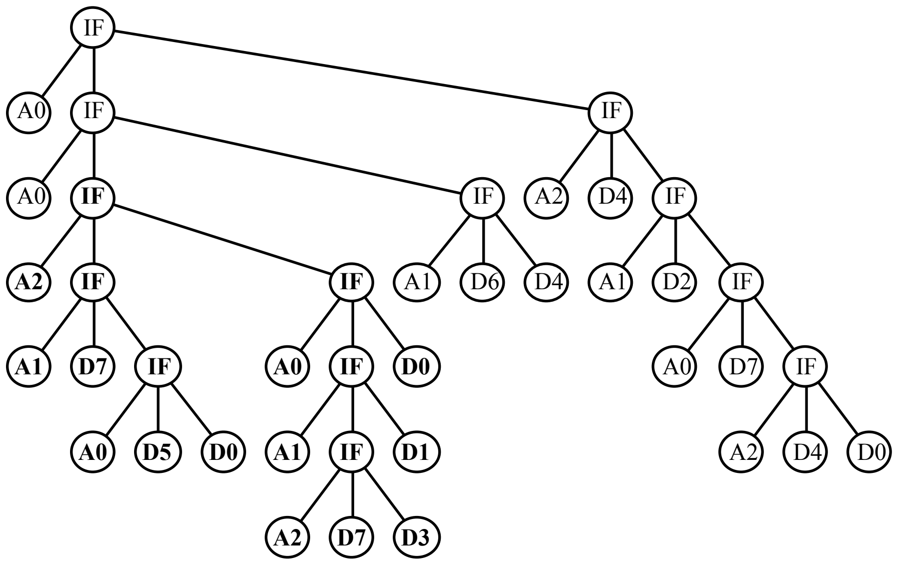


*Figure 1.16 Second parent (scoring 1920 hits) from generation 8 for 100% correct individual in generation 9*

The tree representing this second parent has 40 points. The crossover point chosen at random for this second parent was point 5. This point corresponds to the third occurrence of the function IF. That is, the crossover fragment consists of the emboldened and underlined sub-expression of this second parent.

This sub-expression of this second parent 100% correctly handles the case when A0 is T (i.e. the odd numbered addresses). This sub-expression 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 embedded 6-multiplexer in this second parent 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)
```

In other words, this imperfect second parent handles part of its environment correctly and part of its environment incorrectly. This second parent does not do as well when A0 is NIL (i.e. the even numbered data bits). In other words, this second parent correctly handles the odd-numbered data bits and incorrectly 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.

Figure 1.17 shows this case splitting by showing the 100% correct offspring from generation 9 as two 6-multiplexers:

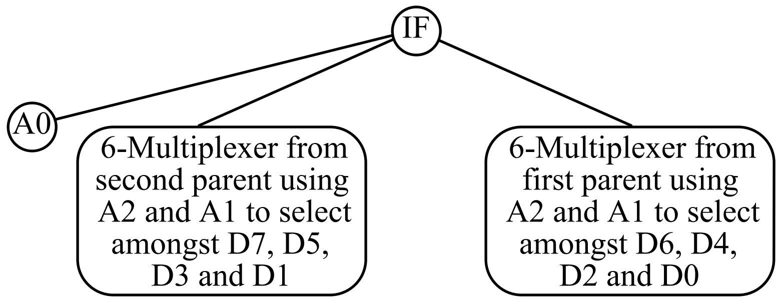

IF
A0
6-Multiplexer from second parent using A2 and A1 to select amongst D7, D5, D3 and D1
6-Multiplexer from first parent using A2 and A1 to select amongst D6, D4, D2 and D0

*Figure 1.17 Simplified 100% correct individual from generation 9 shown as a hierarchy of two 6-multiplexers*

Figure 1.18 also shows this simplified version of the 100% correct individual from generation 9.

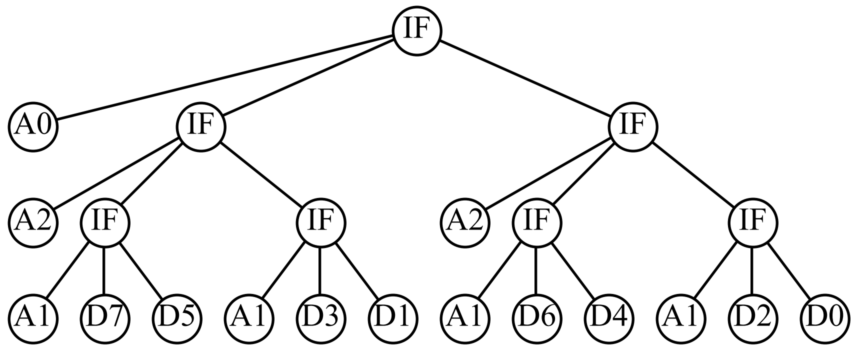


*Figure 1.18 Simplified 100% correct individual from generation 9 shown as a hierarchy of two 6-multiplexers*

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 on local maxima.

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

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. That is, one encounters more S-expressions that perform irrelevant, counterproductive, partially blind, and incorrect work. But if we look at the sequence of hits in the 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.

## 6.2. 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[11], Holland et. al.[12]) because they are often parsimonious and they are a human-like way of dealing with situations. Wilson's[13] noteworthy BOOLE experiments 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[14] modified the credit allocation scheme and successfully produced a default hierarchy.

## 6.3. RESULTS OVER A SERIES OF RUNS

In the previous section we described one particular run of the genetic programming paradigm in which we obtained a solution to the Boolean 11-multiplexer problem.

A basic statistic associated with runs of the genetic programming paradigm (and the genetic algorithm) is the probability of success $p_s$ that a particular run produces a desired result within a specified number of generations $N_{gen}$ with a population of size M. We obtain this statistic $p_s$ by making a substantial number of runs.

Note that this value of $p_s$ depends strongly on the choices of

• the population size M

• the maximum number of generations $N_{gen}$ to be run,

• the secondary parameters of the genetic algorithm, and

• all the fixed minor details of our implementation of the algorithm.

Figure 1.19 shows the probability of success $p_s$ over 200 generations for 309 runs of the Boolean 6-multiplexer problem. This graph rises monotonically since the probability is cumulative from generation 0. This graph approaches 100% asymptotically as the number of generations grows. There is a point after which additional generations produce only small increases in the probability of success $p_s$. For example, in this graph, at generation 51, $p_s$ is about 67%. At generation 101, $p_s$ is 83%. At generation 151, $p_s$ is 87%. At generation 201, $p_s$ is 89%.

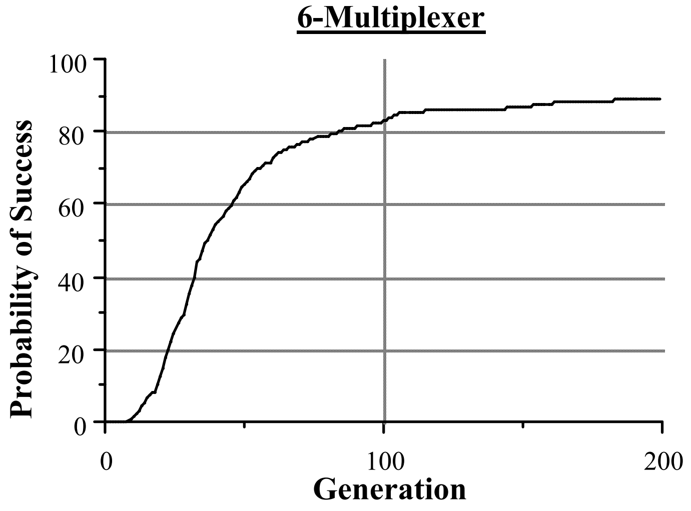


*Figure 1.19 Probability of success for runs of Boolean 6-multiplexer for 200 generations*

Neither conventional genetic algorithms operating on fixed length character strings nor the genetic programming paradigm always produce the desired results on a particular run of the algorithm.

For one thing, genetic algorithms inherently involve probabilistic steps. Because of these probabilistic steps, anything can happen on a given run and nothing is guaranteed. In particular, some runs simply do not produce the desired results within a particular amount of time. For example, genetic algorithms may prematurely converge (i.e. converge to a sub-optimal result). The exponentially increasing allocation of future trials on the basis of the current estimates of the fitness of the population is both the strength and a weakness of genetic algorithms. This allocation is a strength because it is the fundamental reason why genetic algorithms work in the first place. This allocation is a weakness because it may result in premature convergence.

The effects of randomness, premature convergence, unfortuitous initial conditions, and other chaotic affects on genetic algorithms can be minimized by making entirely separate multiple independent runs. These separate multiple independent runs, of course, lend themselves to parallel computer architectures and yield virtually perfect linear speed-up, but that is not the point here. The best single individual from all of these multiple independent runs is then designated as the solution to the problem.

One way to measure the amount of computational resources required by the genetic programming paradigm (or the genetic algorithm in general) is to determine the likely number of independent runs needed to produce a desired result with a certain probability, say, $z = 99\%$. Once we determine the likely number of independent runs required, we can then multiply this number by the amount of processing required for each run. The amount of processing required for each run is generally proportional to the product of the population size and the number of generations executed.

Once we have obtained the probability of success $p_s$ by measurement, we can say that the probability of achieving the desired result at least once within K runs is $1 - (1-p_s)^K$. If we are seeking to achieve the desired result with a probability of, say, $z = 1 - \varepsilon = 99\%$, then the number K of independent runs (niches) required is the result of

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

rounded up to the next highest integer. For example, if $p_S$ is 90%, then K is 2.

We can measure the number of individuals that need to be processed by a genetic algorithm to solve a particular problem. For example, we ran 54 runs of the Boolean 11-multiplexer problem with a population size M of 4,000 and for a maximum number of generations $N_{gen}$ of 201 (i.e. generation 0 plus 200 additional generations).

Figure 1.20 shows the probability of success $p_S$ of a run for various numbers of generations for a population size M of 4000. For example, the graph shows that by generation 10, only about 28% of the runs produced at least one individual with a perfect score of 2048 matches. By generation 15, 78% of the runs produced a perfect solution. By generation 20, 90% of the runs produced a perfect solution.

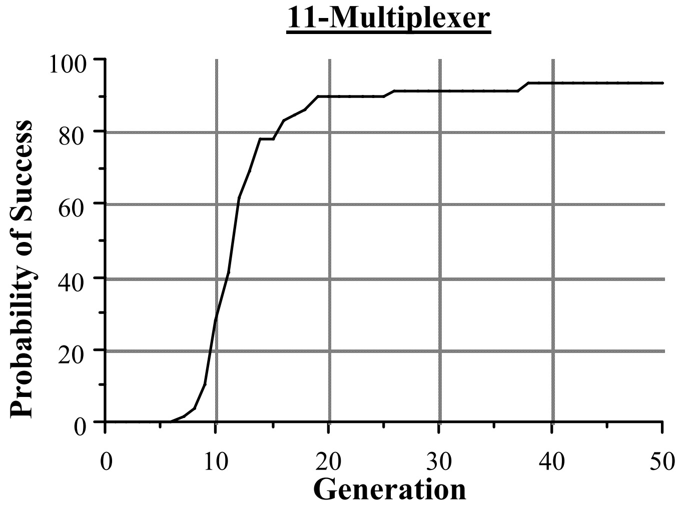


*Figure 1.20 Probability of success for runs of 11-multiplexer problem with population size M of 4000*

With a probability of success $p_S$ = 90% by generation 20, the formula above indicates that K = 2 independent runs are required to assure a 99% probability of solving the problem. Therefore, the number of individuals that need to be processed to assure a 99% probability of solving this problem is no more than 160,000 individuals (i.e. 2 times 20 times 4000). This 160,000 does not reflect the fact that some successful runs would, in actual practice, be terminated without executing all 20 generations. The size of the search space ($2^{2048}$) for the 11-multiplexer is very large in relation to the number of individuals processed that need to be processed (i.e. no more than 160,000).

In the Boolean 11-multiplexer problem described above, we chose an unusually large and decidedly non-optimal population size (i.e. 4000) so as to produce a solution in a sufficiently small number of generations (i.e. 9) to allow us to economically run a genealogical audit trail.

In general, the selection of the optimal population size is a difficult problem for both the genetic programming paradigm and the conventional genetic algorithm operating on strings. The number of individuals that must be processed to give a 99% probability of finding a solution is a complex function of all the factors that influence the probability of success $p_s$. These factors include the population size M, maximum number $N_{gen}$ of generations to be run, the various secondary parameters, and all the other choices (e.g. method of creating the initial population) that are involved in the run.

Figure 1.21, for example, shows the probability of success $p_s$ of a run of the 6-multiplexer problem for various numbers of generations for population sizes of 500, 1000, and 2000. Clearly, the optimal population size for the 6-multiplexer problem is not 500, but is, instead, some larger number in the neighborhood of 1000 to 2000.

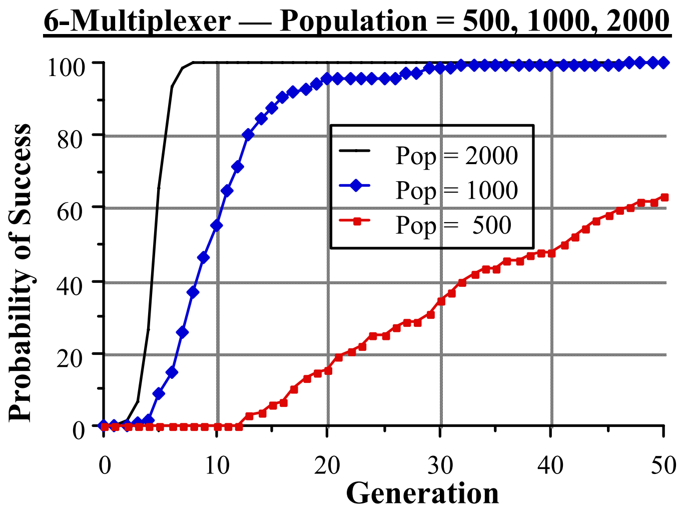


*Figure 1.21 Probability of success for runs of the 6-multiplexer problem for population size M of 500, 1000, and 2000*

## 6.4. 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 5,000,000 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 blind search.

In addition, on the first 1,000,000 random S-expressions, we computed an entire hits histogram of raw fitness values. The high score in this histogram was only 1408 hits (out of a possible 2048) and the low score was 704 hits. Moreover, only 10 individuals achieved this high score of 1408. The high point of the hits histogram distribution came at 1152 hits; the second highest point came at 896 hits; and the third highest point came at 1024 hits. The size of the search space ($2^{2048}$) for the 11-multiplexer is very large in relation to the number of individuals processed in a typical run solving the 11-multiplexer.

A similar control experiment was conducted for the Boolean 6-multiplexer problem (with a search space of $2^{2^6} = 2^{64}$) involving 10,000,000 individuals. As before, no 100% correct individual was found in this blind 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-multiplexer 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, we conclude that *the results described herein are not the fruits of random search*.

As a matter of fact, we have evidence suggesting that the solutions to many functions 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 odd-2-parity function with two Boolean arguments (i.e. exclusive-or function). The odd-2-parity function of k Boolean arguments returns T (True) if the number of arguments equal to T is odd and returns NIL (False) otherwise. There are only $2^{2^2} = 2^4 = 16$ possible Boolean functions with two Boolean arguments and one output. Thus, in the search space of truth tables for 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.

First, we generated 100,000 random individuals using a function set consisting of the three 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 truth tables for Boolean functions of 2 arguments), we would expect about about 6250 in 100,000 (i.e. 1 in 16) random compositions of functions 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 truth tables for Boolean functions.

Second, we generated an additional 100,000 random individuals using a function set consisting of the different function set 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 second 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 truth tables for Boolean functions.

Third, we generated 100,000 random individuals using a function set consisting of four functions taking two arguments each, namely, F = {AND, OR, NAND, NOR}. We found that only 118 out of 100,000 randomly generated compositions realized the exclusive-or function (i.e. a frequency of 1 in 846). That is, with this third function set, randomly generated compositions of functions realizing the exclusive-or function are about 53 times *sparser* than solutions in the original search space of truth tables for Boolean functions.

In other words, solutions to the odd parity (exclusive-or) function with two arguments appear to be 53 to 57 times sparser in the space of randomly generated compositions of functions than solutions in the original search space of the problem.

We then performed similar experiments on two Boolean functions with three Boolean arguments and one output, namely, the odd-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 $2^8 = 256$ 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 odd-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. We cannot make the comparison for the odd-3-parity function, but it is presumably tens of thousands of times scarcer than one in 256.

These three results concerning the odd-3-parity function, the 3-multiplexer function, and the odd-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[13,14], Quinlan[20], Barto et. al.[21]).

In summary, as to these benchmark Boolean functions, 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 the original problem. In Lenat's[22] work on discovering mathematical laws via heuristic search and other related work[23], 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*[22], 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[24]), Lenat recognized that LISP syntax may have overly facilitated discovery of his previously reported results, namely, mathematical laws stated in terms of LISP's list manipulation functions and LISP's primitive object (i.e. the list).

In contrast, the problems described herein are neither stated nor solved in terms of objects or operators unique or peculiar to LISP. The solution to the Boolean multiplexer function is expressed in terms of ordinary Boolean functions (such as AND, OR, NOT, and

IF). The solutions to the numerical problems discussed herein (such as symbolic regression, broom balancing) are expressed in terms of the ordinary arithmetic operations (such as addition, subtraction, multiplication, and division). The solutions to the planning problems (such as block stacking) are expressed in terms of ordinary iteration operations and various domain-specific robotic actions (such as robotic actions that move a block from one place to another).

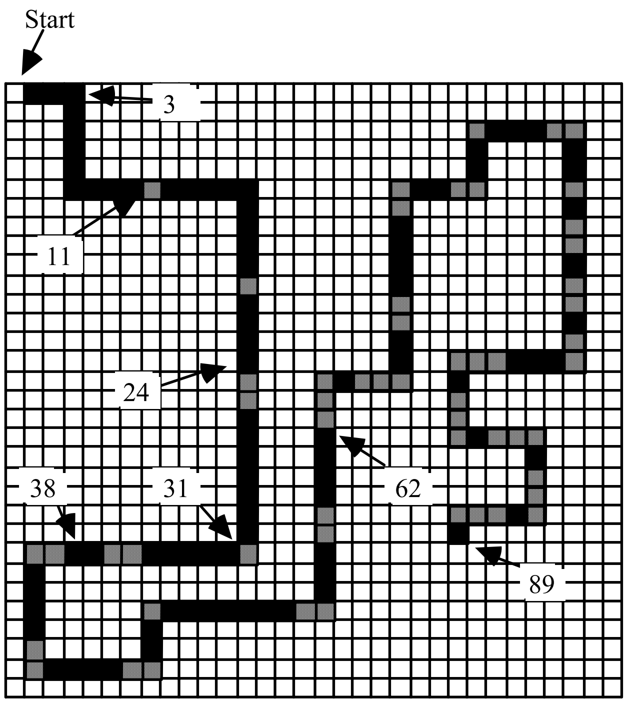
Virtually any programming language could be used to express the solutions to these problems. The LISP programming language was chosen for use in the genetic programming paradigm primarily because of the many convenient features of LISP (most importantly, the fact that data and programs have the same 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 fact, neither lists nor list manipulation functions are involved in any of the problems described herein (except in the irrelevant and indirect sense that the LISP programming language uses lists to do things, unseen by the user, that other programming languages do in different ways).

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 programs generated by the genetic programming paradigm, nor the syntax of the programming language we used to implement the genetic programming paradigm (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. ARTIFICIAL ANT PROBLEM

As a second illustration of the genetic programming paradigm, we consider a task devised by Jefferson et. al.[25] for an artificial ant attempting to find the food lying along an irregular trail.

The setting for the problem is a square 32 by 32 toroidal grid in the plane. The John Muir trail (and the somewhat more difficult Santa Fe trail designed by Christopher Langton) is an irregular winding trail with food in 89 of the 1024 cells. The Santa Fe rail has single gaps,



double gaps, single gaps at corners, double gaps (knight moves) at corners, and triple gaps (long knight moves) at corners. The artificial ant begins in the cell identified by the coordinates (0,0) and is facing in a designated direction (i.e. east).

The Santa Fe trail is shown in Figure 1.22. Food is represented by solid black squares, while gaps in the trail are represented by gray squares. The numbers identify key features of the trail in terms of the number of pieces of food occurring up to that feature. For example, the number 3 highlights the first corner. It appears after 3 pieces of food. Similarly, the number 11 highlights the first gap in the trail. The number 38 highlights the first knight's move.

*Figure 1.22 Santa Fe trail for the artificial ant problem with the 89 pieces of food shown in black*

The goal of this problem is to find a computer program for performing the task of following the trail and eating all of the food.

The artificial ant has a very narrow and limited view of the world. In particular, the ant has a sensor that can see only the single immediately adjacent cell in the direction the ant is currently facing. In addition, the ant is limited to three very simple, local actions. Specifically, at each time step, the ant can execute one of following four functions:

- RIGHT turns the ant right (and does not move the ant).
- LEFT turns the ant left (and does not move the ant).

- MOVE moves the ant forward in the direction it is facing. When an ant moves into a square, it eats the food, if any, in that square (thus eliminating food from that square).
- IF-FOOD-HERE senses the contents of the single immediately adjacent cell in the direction the ant is facing and allows one of two alternative actions to be taken based on whether food is present.

The ant's goal is to traverse the entire trail and collect all of the food within a reasonable limited number of time steps.

When Jefferson et. al. used the conventional genetic algorithm operating on strings to find the finite state automaton to solve this problem, it was first necessary to develop a representation scheme to convert the state transition table of the potential automaton into binary strings of length 453. In the genetic programming paradigm, the problem can be approached and solved in a more natural and direct way.

The first and second major steps in using the genetic programming paradigm are to identify the set of terminals and functions. We adopt the four operators defined and used by Jefferson, namely, IF-FOOD-HERE, MOVE, RIGHT, and LEFT.

In this problem, we are not primarily concerned with the three overt state variables of the ant (i.e. the vertical and horizontal position of the ant on the grid and the direction the ant is facing). Instead, we are concerned with finding food. And, to find food, we must make use of the information that the ant's very limited sensor provides about food in the outside world. In this problem, the information we want to process is the information coming in from the outside world via the ant's sensor. Thus, one natural approach to this problem using the genetic programming paradigm is to put the sensing function IF-FOOD-HERE into the function set. The IF-FOOD-HERE function has two arguments and executes the first argument if the ant’s sensor senses food, or, otherwise, executes the second argument.

If the function set for this problem contains the one operation that processes information, the terminal set should then contain the actions which the ant should take given the outcomes of this information processing. Thus, the terminal set for this problem is

```
T = {MOVE, RIGHT, LEFT}.
```

These three terminals are actually functions that operate via their side effects on the ant’s state (i.e. the ant's horizontal and vertical position on the grid and the ant's facing direction). These terminals are functions with no arguments.

The IF-FOOD-HERE function is the only essential function for the function set of this problem; however, it is often useful to include some connective glue in the function set to facilitate the formation of sequences of operations.

The PROGN function is the Common LISP connective function that sequentially executes its arguments from left to right as individual steps in a program. For example, the 2-step PROGN function

```
(PROGN (RIGHT) (LEFT))
```

turns the ant to the right and then turns the ant to the left. It is often useful to include the PROGN function in the function set with both two and three arguments.

Thus, the function set for this problem is

```
F = {IF-FOOD-HERE, PROGN, PROGN}
```

taking 2, 2, and 3 arguments, respectively.

The third major step in using the genetic programming paradigm is to identify the fitness function. The natural measure of fitness of a given computer program in this problem is the amount of food found by an ant executing the given program. We allowed the ant 400 time steps for a given program. Thus, the raw fitness of a computer program for this problem is the amount of food (ranging from 0 to 89) that the ant has found within the maximum allowed amount of time.

For this problem, a bigger value of raw fitness (i.e. amount of food eaten) is better. Thus, standardized fitness for this problem is the maximum value of raw fitness (i.e. 89) minus raw fitness.

Note that there are no explicit fitness cases in this problem. The implicit fitness cases are the various states of the ant (i.e. its position and facing direction) that arise along the ant's actual trajectory. These are sufficiently representative for this particular problem to allow the ant to learn to solve this problem.

The genetic programming paradigm starts with the generation of 500 random computer programs recursively composed from the available functions and terminals.

Predictably, this initial population of random computer programs includes a wide variety of highly unfit computer programs. The most common type of individual in the initial random population for this problem fails to move at all. For example, the computer program

```
(PROGN (RIGHT) (LEFT))
```

unconditionally turns the ant right and left while not moving the ant anywhere. Similarly, the program

```
(IF-FOOD-HERE (RIGHT) (LEFT))
```

senses some information from the outside world and then conditionally turns the ant various ways while still not moving. Both of these highly unfit individuals get 0 of the 89 pieces of food when they are mercifully terminated by the expiration of the allotted time.

Some randomly generated computer programs move without turning. For example:

```
(MOVE)
```

shoots across the grid from west to east without either looking or turning. This highly active, albeit undirected, behavior finds 3 of the 89 pieces of food.

Another highly unfit random computer program (which we call the "quilter" because it traces a quilt-like pattern across the toroidal grid) moves and turns without looking.

```
(PROGN (RIGHT)
       (PROGN (MOVE) (MOVE) (MOVE))
       (PROGN (LEFT) (MOVE)))
```

Another randomly generated computer program (which we call the "looper") finds the first 11 pieces of food on the trail and then goes into an infinite loop when it encounters the first single gap in the trail. One randomly generated computer program (which we call the "avoider") actually correctly takes note of some of the food along the trail until the first gap in the trail. Then, it actively avoids this food by carefully moving around it until it eventually returns to its starting point. The S-expression for the avoider is

```
(IF-FOOD-HERE (RIGHT)
              (IF-FOOD-HERE (RIGHT)
                            (PROGN (MOVE) (LEFT)).
```

The avoider's path is marked by X's in Figure 1.23.

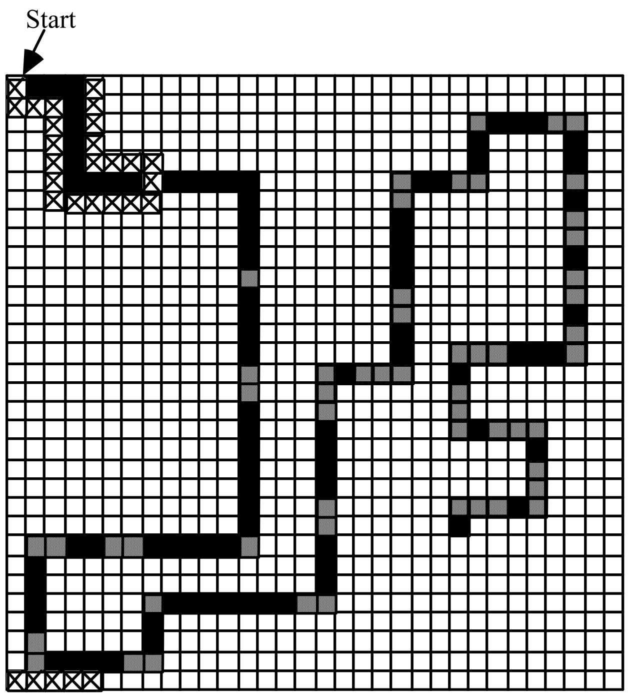


*Figure 1.23 Avoider's path along Santa Fe trail*

In one run, the best single individual in the initial random population was able to find 32 of the 89 pieces of food, whereas the worst single individual in the population found none of the food. The average amount of food found was about 3.5 pieces.

The Darwinian reproduction operation and the genetic crossover operation were then applied to parents selected from the current population with probabilities proportionate to fitness to breed a new population of offspring computer programs. Although the vast majority of the new offspring computer programs are again highly unfit, some of them tend to be somewhat more fit than others. Moreover, over a period of time and many generations, some of them tend to be slightly more fit than those existing in earlier generations.

Figure 1.24 shows the standardized fitness of the worst single individual, the standardized fitness of the best single individual, and the average standardized fitness of the population between generations 0 and 21 of one particular run of the artificial ant. As can be seen, the standardized fitness of the best single individual generally improves (i.e. trends towards zero) from generation to generation, although this improvement is not monotonic. The average value of standardized fitness value starts at about 85.5 (i.e. 3.5 pieces of food found) and then generally improves from generation to generation. Note that there is at least one individual in the population at every generation that finds no food at all so that the worst-of-generation plot runs horizontally across the top of the graph with a fitness value of 89 (i.e. zero pieces of food).

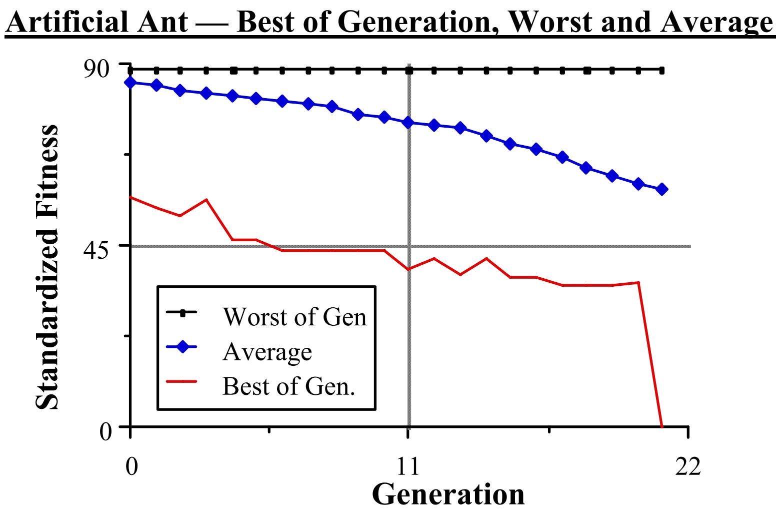


*Figure 1.24 Standardized fitness of worst-of-generation individual, average standardized fitness of population, and standardized fitness of best-of-generation individual for the artificial ant problem.*

The individual computer program scoring 89 out of 89 that emerged on generation 21 is shown below:

```
(IF-FOOD-HERE (MOVE)
              (PROGN (LEFT)
                     (PROGN (IF-FOOD-HERE (MOVE)
                                          (RIGHT))
                            (PROGN (RIGHT)
                                   (PROGN (LEFT)
                                          (RIGHT))))
                     (PROGN (IF-FOOD-HERE (MOVE)
                                          (LEFT))
                            (MOVE))))
```

This individual S-expression has 18 points and is graphically depicted in Figure 1.25.

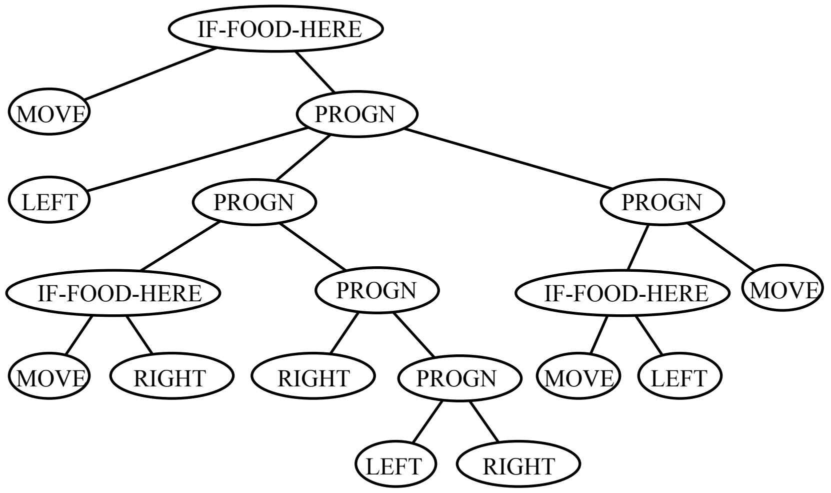


*Figure 1.25 Solution to artificial ant problem from generation 21*

This individual LISP S-expression is a 100% correct solution to this problem. The interpretation of this S-expression is as follows: The test IF-FOOD-HERE senses whether there is any food in the square that the ant is facing. If food is present, the left branch of the IF-FOOD-HERE test is executed and the ant MOVES forward. When the ant moves onto a place on the grid with food, the food is eaten and the ant receives credit for the food.

If the IF-FOOD-HERE test at the beginning of the S-expression senses no food, the ant enters the 3-step PROGN sequence immediately below the IF-FOOD-HERE test. The ant first turns LEFT. Then, a 2-step PROGN sequence begins with the test IF-FOOD-HERE. If food is present, the ant MOVES forward. If not, the ant turns RIGHT. Then, the ant turns RIGHT again. Then, the ant pointlessly turns LEFT and RIGHT in another 2-step PROGN sequence. The net effect is that the ant is now facing right relative to its initial facing direction. The ant next executes the final 2-step PROGN subtree at the far right of the figure. If the ant now senses food via the IF-FOOD-HERE test, the ant MOVES forward. Otherwise, the ant turns LEFT. The ant has now returned to its initial facing direction. The ant now unconditionally MOVES. Note that there is no testing of the backwards directions. The repeated application of this control program allows the ant to negotiate all of the gaps and irregularities of the trail and to collect all of the food in the allotted time.

In summary, we have shown how to use the genetic programming paradigm to genetically breed a computer program that successfully navigates the artificial ant so as to find 100% of the food along the Santa Fe trail.

Note that in the genetic programming paradigm, we made no assumption in advance about the size, shape, or complexity of the eventual solution. The solution found above in generation 21 had 18 points. We did not specify that the solution would have 18 points nor did we specify the shape or content of the S-expression. The size, shape, and content of the S-expression that solves this problem evolved in response to the selective pressure provided by the fitness measure (i.e. amount of food eaten).

Figure 1.26 shows the probability of success $p_S$ (computed from 55 runs) that at least one S-expression causes the artificial ant to traverse the entire trail and collect all 89 pieces of food (before timing out) as a function of the number of generations for population sizes of 500, 1000, and 2000. In particular, the probability of success $p_S$ by generation 10 with a population size of 2000 is 40%. With $p_S = 40\%$, K is 9, and no more than 180,000 individuals (i.e. 9 times 2000 times 10) need to be processed to assure a 99% probability of solving the problem.

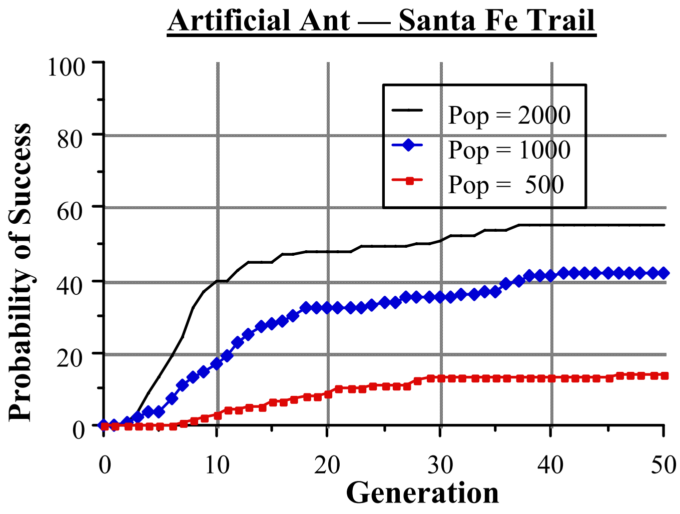


*Figure 1.26 Probability of success for the artificial ant problem for population sizes of 500, 1000, and 2000*

## 8. SOLVING A PAIR OF LINEAR EQUATIONS

As a third illustration of the genetic programming paradigm, consider the problem of finding a formula for solving a pair of linear equations.

In particular, suppose we want to solve a pair of consistent, non-indeterminate linear equations

$$a_{11}x_1 + a_{12}x_2 = b_1$$
$$a_{21}x_1 + a_{22}x_2 = b_2$$

for the first of its two real-valued variables ($x_1$). In other words, we are seeking a computer program that takes $a_{11}$, $a_{12}$, $a_{21}$, $a_{22}$, $b_1$, and $b_2$ as its inputs and produces $x_1$ as its output. Without loss of generality, we can assume that the coefficients of the equations were prenormalized so the determinant is one. The solution to this problem can be viewed as a search for a mathematical expression (S-expression) from a hyperspace of possible mathematical expressions that can be composed from a set of available functions and arguments.

The first major step in using the genetic programming paradigm is to identify the set of terminals. In the previous problems, the terminal set consisted of the information which the mathematical expression must process in order to solve the problem. In this problem, the information that must be processed by a computer program to find $x_1$ are the values of $a_{11}$, $a_{12}$, $a_{21}$, $a_{22}$, $b_1$, and $b_2$. Thus, the terminal set is

```
T = {A11, A12, A21, A22, B1, B2}.
```

The second major step in using the genetic programming paradigm is to identify the set of functions. The set of functions that are used to generate the mathematical expressions that attempt to fit the given finite sample of data. The function set for this problem might consist of addition (+), subtraction (-), multiplication (*), and the protected division function (%) described previously. Thus, the function set is

```
F = {+, -, *,%}.
```

Each of these four functions takes two arguments.

The third major step in using the genetic programming paradigm is to identify the fitness function. The fitness cases that will be used to evaluate the fitness of any proposed S-expression are 10 randomly generated pairs of consistent, non-indeterminate linear equations (i.e. set of values of $a_{11}$, $a_{12}$, $a_{21}$, $a_{22}$, $b_1$, and $b_2$ and the associated correct values of $x_1$ and $x_2$). In this problem, the raw fitness is measured by the erroneousness of the S-expression. Each genetically produced S-expression is evaluated for fitness in the following way: First, the value of the first unknown variable $x_1$ produced by the genetically produced S-expression as the solution to equation pair i (i.e. $x^g{}_{1i}$) is substituted into one equation of the pair i to find the corresponding value of the second unknown variable $x_2$ (i.e. $x^g{}_{2i}$). Second, we determine the Euclidean distance in the plane between the genetically produced solution point ($x^g{}_{1i}$, $x^g{}_{2i}$) for equation pair i and the actual solution point ($x^s{}_{1i}$, $x^s{}_{2i}$) for equation pair i. Third, these distances are summed over all 10 pairs of equations. The sum of these distances is the raw fitness of the S-expression. If the S-expression were a correct general formula for solving a pair of linear equations, the sum of these distances would be zero. Thus, standardized fitness equals raw fitness for this problem.

The auxiliary hits measure is defined such that we score one hit if the distance between the genetically produced solution point and the actual solution point for a particular pair of equations is less than .01.

Figure 1.27 shows, for i of 1 and 2 only, the distance (error) between the solution point ($x^g{}_{1i}$, $x^g{}_{2i}$) produced by a genetically produced S-expression and the actual solution point ($x^s{}_{1i}$, $x^s{}_{2i}$) for equation pair i. The lines in the figure connect the two points applying to a given equation pair. As such, the lines graphically represent the error. If the S-expression were the correct general formula for solving a pair of linear equations, the two points would overlap and there would be no line shown (i.e. the error would be zero).

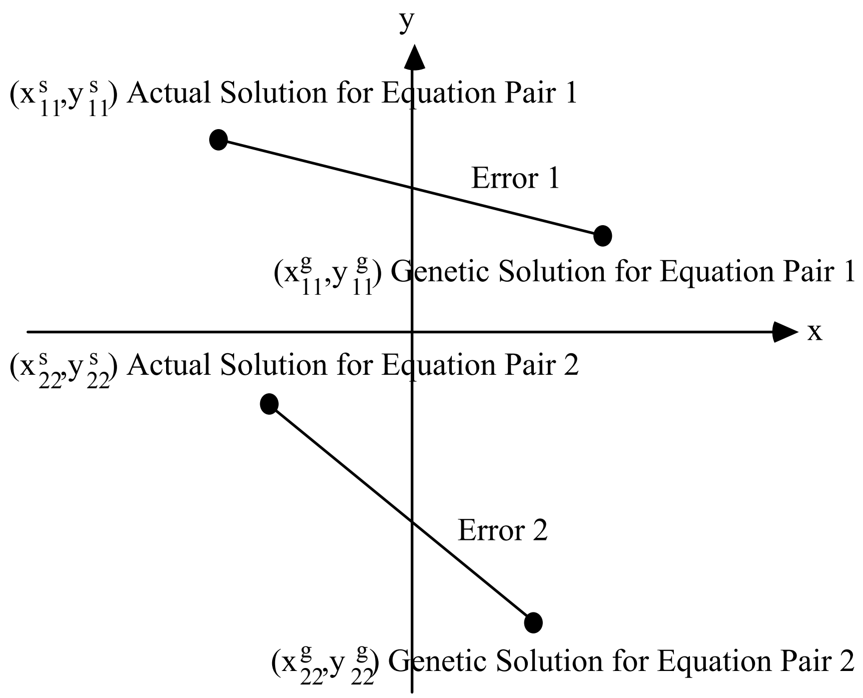

y
$(x^s_{11}, y^s_{11})$ Actual Solution for Equation Pair 1
Error 1
$(x^g_{11}, y^g_{11})$ Genetic Solution for Equation Pair 1
x
$(x^s_{22}, y^s_{22})$ Actual Solution for Equation Pair 2
Error 2
$(x^g_{22}, y^g_{22})$ Genetic Solution for Equation Pair 2

*Figure 1.27 The line in the top half of this figure connects the actual solution point for equation pair 1 with the genetically produced solution point for equation pair 1. This line represents the error associated with equation pair 1. The line in the bottom half of this figure represents for error for equation pair 2.*

Predictably, this initial population of random S-expressions includes a wide variety of highly unfit S-expressions.

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

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

The Darwinian reproduction operation and the genetic crossover operation are then applied to parents selected from the current population with probabilities proportionate to fitness to breed a new population of offspring computer programs. Although the vast majority of the new offspring computer programs are again highly unfit, some of them tend to be somewhat more fit than others. Moreover, over a period of time and many generations, some of them tend to be slightly more fit than those existing in earlier generations.

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 single best individual S-expression from generation 0 had a raw fitness value of 125.8 and is shown below:

```
(+ (- A12 (* A12 B2)) (+ (* A12 B1) B2)).
```

This S-expression is equivalent to

$$a_{12}b_1 + b_2 + a_{12} - a_{12}b_2$$

The best individual begins improving and has a fitness 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 best single S-expression in generations 21 and 22 had a fitness value of 62 and is shown below:

```
(+ (- A12 (* A12 B2)) (* A22 B1))
```

This S-expression is equivalent to

$$a_{22}b_1 + a_{12} - a_{12}b_2$$

This individual differed from the known correct solution only by one term, namely + A12.

The best single S-expression in generations 23 through 26 had a raw fitness value of 58 and is shown below:

```
(+ (- A22 (* A12 B2)) (* A22 B1))
```

This S-expression is equivalent to

$$a_{22}b_1 + a_{22} - a_{12}b_2$$

This individual differed from the known correct solution only by one term, namely + A22.

Starting with generation 27, a perfect solution for $x_1$ emerges, namely

```
(- (* A22 B1) (* A12 B2)).
```

This S-expression is equivalent to

$$a_{22}b_1 - a_{12}b_2.$$

Figure 1.28 shows the probability of success $p_s$ (based on 122 runs) that at least one S-expression scores 10 hits (i.e. the error is less than .01 for all 10 pairs of equations) as a function of the number of generations for a population size M = 500. Since the probability of success $p_s$ by generation 15 is 44%, K is 8.

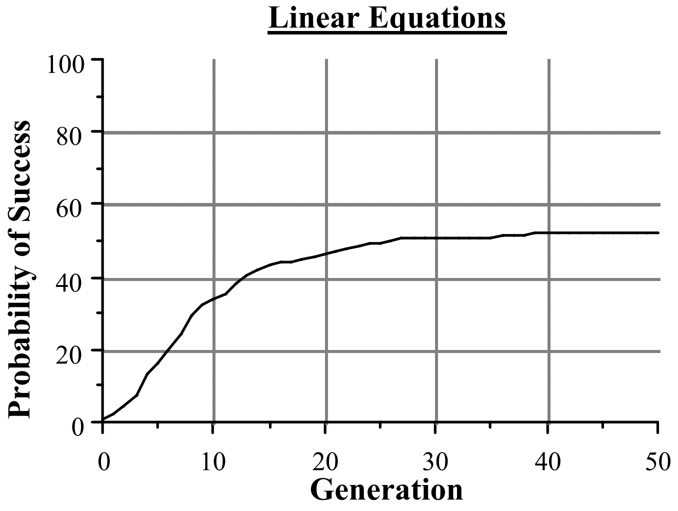


*Figure 1.28 Probability of success for the linear equations problem for a population size of 500*

## 9. RANDOMIZER

The problem of writing a computer program that generates a stream of pseudo-random numbers illustrates another way of measuring fitness.

Numbers "chosen at random" are useful in a variety of scientific, mathematical, engineering, and industrial applications, including Monte Carlo simulations, sampling, decision theory, game theory, instant lottery ticket production, etc. However, random numbers are difficult to create.

Our goal is to genetically breed a computer program to convert a sequence of consecutive integers into a sequence of random binary digits. The input to our randomizer will merely be an argument J running consecutively from 1 to 16,384 ($2^{14}$). In other words, each random binary digit as output will be a function of a consecutive integer J as input.

The first major step in using the genetic programming paradigm is to identify the terminal set. The set of terminals (along with the set of functions) are the ingredients from which the S-expressions are composed. The only variable in each S-expression for a randomizer is the argument J. The terminal set for this problem also contains several small integers so that

```
T = {J, 0, 1, 2, 3}.
```

The second major step in using the genetic programming paradigm is to identify the function set. Since we anticipate creation of a randomizer consisting of steps similar to those used in congruential randomizers, the function set for this problem is

```
F = {+, -, *, QUOT%, MOD%}
```

taking two arguments each. The protected modulus function MOD% uses the protected division function % in computing the modulus. The protected integer quotient function QUOT% uses the protected division function % in computing the integer quotient.

An S-expression composed of the above functions and terminals will always produce a numerical value. Since we want binary digits as output, we wrap the S-expression in an output interface (which we call the wrapper) which specifies that any positive numerical output will be interpreted as a binary one while any other output will be interpreted as a binary zero.

The third major step in using the genetic programming paradigm is to identify the fitness function. This problem has l6,384 fitness cases, namely, the values of J ranging between 0 and 16,383. The fitness measure for this problem will be a number computed from the entire sequence of 16,384 binary digits. There are numerous possible approaches to measuring the randomness of sequences (Knuth[27]). Our goal is to have statistical independence among the sequence of binary digits. In particular, we desire that, for any integer N (where N runs from 1 to infinity), the probabilities of each of the $2^N$ possible sub-sequences of length N should all be equal to $\frac{1}{2^N}$ (within an acceptably small error $\varepsilon \geq 0$) . No finite sequence can satisfy the above test. However, if the window size N is then limited to some finite fixed integer $N_{max}$, then “only” $2^{N_{max}}$ probabilities must be estimated when $N = N_{max}$.

More importantly, these $2^{N_{max}}$ separate probabilities can be conveniently summarized into a scalar quantity by using the concept of entropy for this set of events and probabilities. The entropy (which is measured in bits) is maximal when the probabilities of all the possible events are equal. The entropy $E_h$ for the set of $2^h$ probabilities for the $2^h$ possible sub-sequences of length h, equals

$$E_h = -\sum_j P_{hj} \log_2 P_{hj} .$$

The index j in this summation ranges over the $2^h$ possible sub-sequences of length h. By convention, $\log_2 0$ is 0 when computing entropy. This sum attains its maximum value of h precisely when the probabilities of all the $2^h$ possible sub-sequences of length h are equal to $\frac{1}{2^h}$.

As h runs from 1 to $N_{max}$, it is convenient to further summarize the $N_{max}$ separate scalar values of entropy into a single scalar value by summing them to obtain $E_{total}$ as follows:

$$E_{total} = \sum_{h=1}^{N_{max}} \left[ -\sum_j P_{hj} \log_2 P_{hj} \right]$$

When $E_{total}$ attains the maximal value of

$$\sum_{h=1}^{N_{max}} h = N_{max}(N_{max} - 1),$$

then the sequence may be viewed as being random (in this sense).

If we choose $N_{max} = 7$, then the maximum raw fitness associated with the best result will be 28 bits. Standardized fitness is 28 minus raw fitness.

In one particular run, the best single S-expression of the 500 individuals in the initial random generation scored 20.920 bits. This S-expression consisted of 63 points. When simplified, this best-of-generation individual is equivalent to

```
(+ J (* (* (MOD% J 3) 3) (QUOT% (+ J 1) 4))).
```

This best-of-generation individual does a credible job of randomizing bits when the window is narrow. In particular, it gets a perfect 1.000 bits out of a possible 1.000 bits for sub-sequences of length 1, and it gets 1.918 out of a possible 2.000 bits for sub-sequences of length 2. In contrast, this best-of-generation individual gets only 4.002 bits out of a possible 6.000 for sub-sequences of length 6 (i.e. only 67% of the possible score), and it gets only 4.252

bits out of a possible 7.000 for sub-sequences of length 7 (i.e. only 61% of the possible score).

In Figure 1.29, the horizontal axis ranges over the $2^7 = 128$ possible sub-sequences of length 7. The vertical axis is the number of occurrences of each of the 128 possible sub-sequences for the best-of-generation individual for generation 0 for 16,384 values of J. A maximal entropy randomizer would have $\frac{16,384}{128} = 128$ occurrences of each of the 128 possible sub-sequences for 16,384 values of J. As can be seen, the best-of-generation individual from the initial random generation has about 1365 occurrences each for 12 of the 128 possible sub-sequences of length 7.

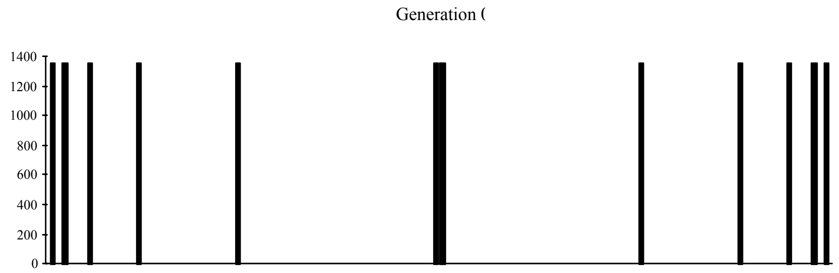


*Figure 1.29 Best-of-generation randomizer for generation 0 showing the frequencies of the 128 different 7 bit patterns*

After 2 generations of this run, the entropy of the best-of-generation individual improved to 22.126 bits.

After 4 generations, the entropy of the best-of-generation individual improved to 26.474 bits.

Figure 1.30 shows the best-of-generation individual for generation 4. As can be seen, after only 4 generations, many more of the 128 possible sub-sequences of length 7 are now represented.

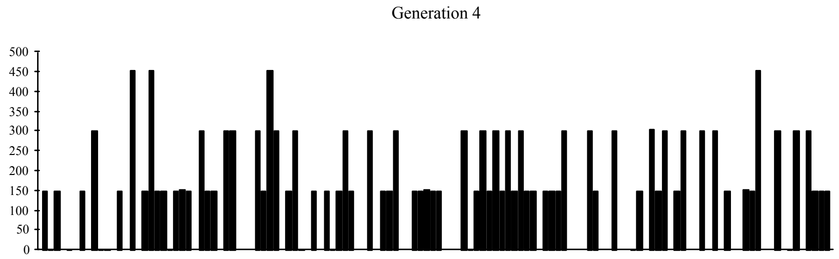


*Figure 1.30 Best-of-generation randomizer for generation 4 showing the frequencies of the 128 different 7 bit patterns*

Figure 1.31 shows the progress, from generation to generation, of each of the 7 components (i.e. for h = 1 through 7) of $E_{total}$ for the best-of-generation individual for generations 0 through 14. As can be seen, entropy for short sub-sequence lengths reaches its maximum level after just a few generations, while entropy for the longer sub-sequence lengths requires additional generations.

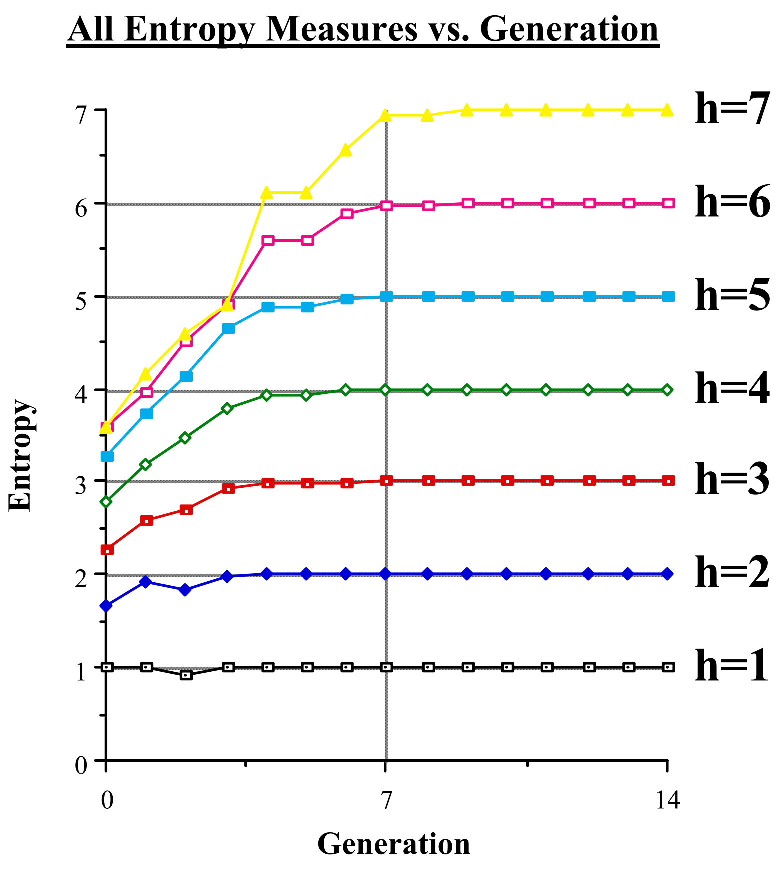


*Figure 1.31 The seven components of total entropy $E_{total}$ for the best-of-generation individual for generations 0 through 14*

Between generations 5 and 13, entropy attained and slowly improved within the 27.800 to 27.900 area.

Figure 1.32 shows that by generation 7 all 128 sub-sequences of length 7 are generated by the best-of-generation randomizer. The number of occurrences are, however, far from equal at this stage.

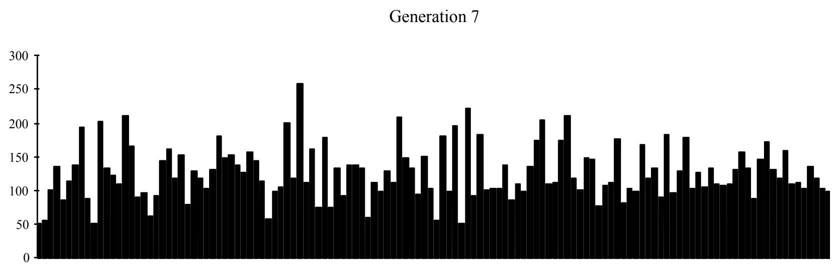


*Figure 1.32 Best-of-generation randomizer for generation 7 showing the frequencies of the 128 different 7 bit patterns*

On generation 14, we obtained an individual S-expression that attained a nearly maximal entropy of 27.996. This S-expression has 153 points, but simplifies to the following individual with only 87 points:

```
(- J (QUOT% (+ (+ (+ J J) J) (* (+ J 2) J)) (+ (MOD%
(* (- 2 1) (QUOT% (QUOT% (+ (* J J) (QUOT% (- (QUOT%
(* J (MOD% (QUOT% J 3) (MOD% J J))) (QUOT% (* 3 2)
(QUOT% 2 1))) (- 3 (QUOT% (+ (* J J) (- 2 1)) 3))) (*
3 (+ (MOD% 1 0) J)))) 3) 3)) (+ (- 2 J) 1)) (+ (QUOT%
(MOD% J 3) (- (MOD% 2 0) (MOD% (MOD% 0 J) J))) (- 3
3)))))
```

Figure 1.33 shows the simplified version (with 41 points) of the best-of-generation individual from generation 14 (with entropy of 27.996) for the randomizer problem.

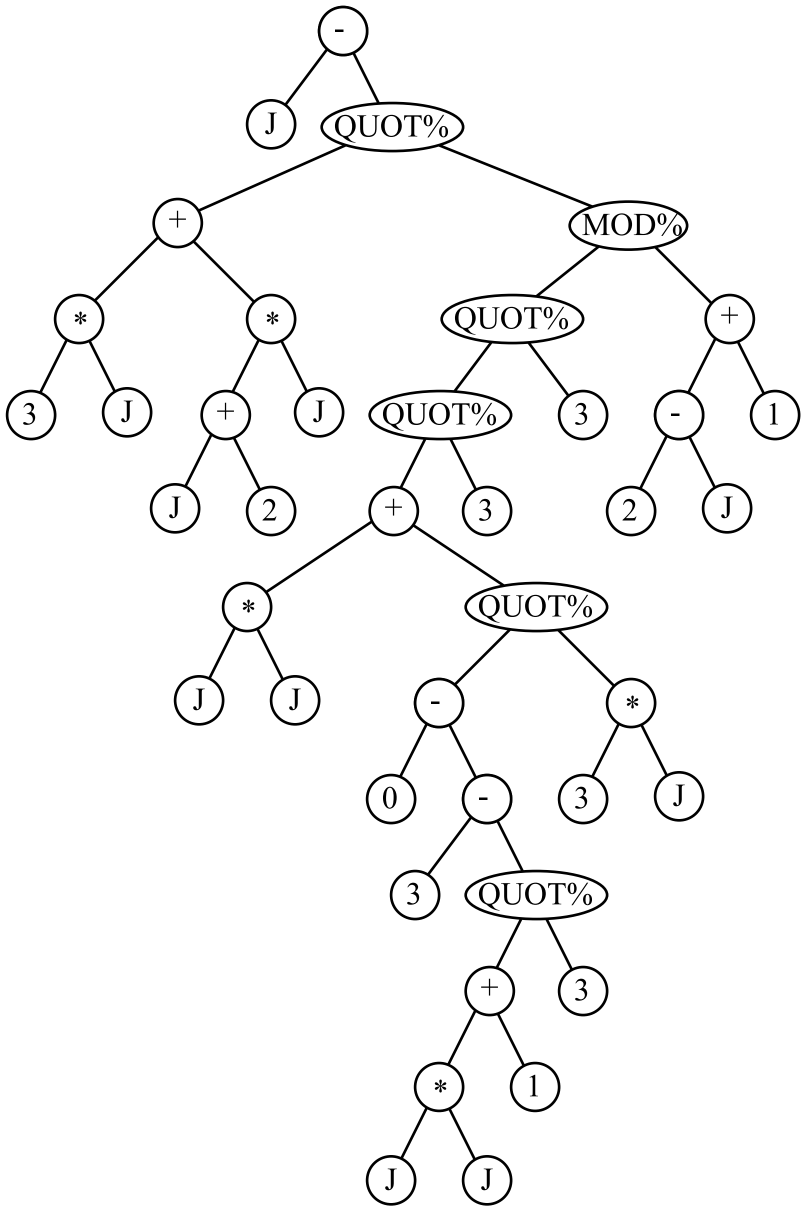


*Figure 1.33 Simplified version of best-of-generation individual from generation 14 of randomizer problem (with entropy of 27.996)*

In scoring 27.996, this randomizer achieved a maximal value of entropy of 1.000, 2.000, 3.000, 4.000, 5.000, and 6.000 bits for sequences of lengths 1, 2, 3, 4, 5, and 6, respectively, and a near-maximal value of 6.996 for the 128 ($2^7$) possible sequences of length 7.

Figure 1.34 shows that each of the 128 possible sub-sequences.of length 7 are generated by the best-of-generation individual from generation 14 (with entropy of 27.996). The number of occurrences of each sub-sequence is in the neighborhood of 128.

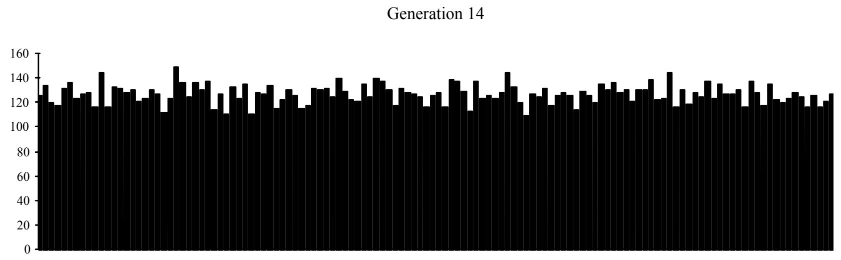


*Figure 1.34 Best-of-generation randomizer for generation 14 showing the frequencies of the 128 different 7 bit patterns*

Note that the progressive change in size and shape of the individuals in the population is a characteristic of the genetic programming paradigm. The size (153 points) and shape of the best scoring individual from generation 14 differs from the size (63 points) and shape of the best scoring individual from generation 0. The size and particular hierarchical structure of the best scoring individual from generation 14 was not specified in advance. Instead, the entire structure evolved as a result of reproduction, crossover, and the relentless pressure of the fitness measure (i.e. entropy). Note that achieving better entropy requires a more complex computation.

Figure 1.35 shows the probability of success $p_S$ (based on 10 runs) of a run with a population size of M = 500 and with success defined as attaining entropy of 27.990 or better. The probability of success $p_S$ = 0.90 by generation 15. Thus, in order to assure a 99% probability of solving the problem, we need K = 2 independent runs with a population of 500 for 15 generations. That is, no more than 15,000 individuals need be processed.

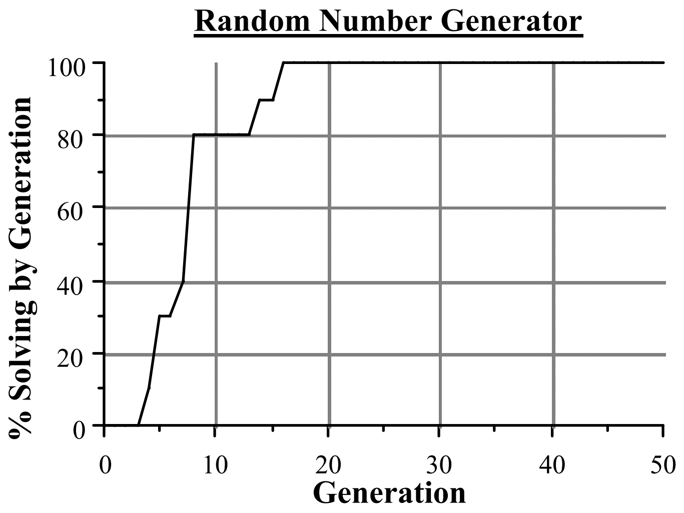


*Figure 1.35 Probability of success for randomizer problem with population size of 500*

We now compare the genetically produced randomizer with the following five randomizers:

- The Park-Miller congruential randomizer described in Anderson[26]

  $x_i = 7^5 x_{i-1} \mod [2^{31} - 1].$

- IBM's URN08 (RANDU) congruential randomizer

  $x_i = 65539 x_{i-1} \mod 2^{31}.$

- The SR[3,28,31] shift register randomizer starting with a seed value $x_0$ and then producing subsequent elements of the random sequence recursively in a shift register (end off, with zero fill) in a 31-bit shift register.

```
    temp =(XOR xi-1 (SHIFT-RIGHT xi 3))
    xi =(XOR temp (SHIFT-LEFT temp 28).
```

where XOR is the exclusive-or operation.

- The two-sequence shuffling randomizer SHUFFLE using the Park-Miller multiplicative congruential randomizer to produce an initial set of uniformly distributed random numbers between 0.0 and 1.0 and then using the shift register randomizer SR[3,28,31] to call out particular numbers from this set of numbers and additional calls on the Park-Miller randomizer to replace the numbers called out .

- The RANDOM randomizer from Texas Instruments.

The table below compares the shortfall in entropy from the maximal 28.000 bits for the genetically bred randomizer and the five commercial randomizers described earlier. We used the same 16,384 points and look-back of h = 7.

| **Randomizer** | **Entropy Shortfall** |
|---|---|
| Park-Miller | .009 |
| IBM RANDU | .010 |
| Shift-Register | .010 |
| SHUFFLE | .015 |
| TI RANDOM | .009 |
| Genetic | .004 |

As can be seen, the genetically bred randomizer has precisely the characteristic for which it was bred (i.e. high entropy). With respect to that particular measure of randomness, it exceeded the performance of the other five randomizers.

## 10. SEQUENCE INDUCTION

Sequence induction involves discovering a mathematical expression (computer program, LISP S-expression) that can generate any arbitrary element in an infinite sequence

$$S = S_0,S_1,...,S_j,...$$

after seeing only a relatively small finite number of specific examples of the values of the sequence.

For example, suppose one is given

$$S = 1, 2, 5, 10, 17, 26, 37, 50, 65, ...$$

as the first nine values of an unknown sequence. If the index j starts at zero, one would easily induce the computational procedure $j^2 + 1$ as the way to compute the sequence element $S_j$ for any specified index j.

Of course, there is no one correct answer to an induction problem. There are an infinity of sequences which agree with the finite number of specific examples in the given sequence. Nonetheless, induction is at the heart of learning and the ability to correctly perform induction is widely viewed as an important component of human intelligence.

Consider the following example of sequence induction. Suppose one is given the first 20 values of the simple non-recursive sequence of integers

S = 1, 15, 129, 547, 1593, 3711, 7465, 13539, 22737,
35983, 54321, 78915, 111049, 152127, 203673,
267331, 344865, 438159, 549217, 680163, ...

The goal is to identify a mathematical expression that produces this sequence of integers.

Sequence induction is symbolic regression (symbolic function identification) where the domain (i.e. independent variable) ranges over the integers 0, 1, 2, and 3....

The terminal set for this problem consists of the index J (i.e. the independent variable) and small integers such as 0, 1, 2, and 3. That is,

```
T = {J, 0, 1, 2, 3}
```

The function set for this problem is

```
F = {+, -, *}.
```

The fitness cases for this problem consist of the first 20 elements of the given sequence. Raw fitness is the sum, taken over the 20 fitness cases, of the absolute value of the difference between the value produced by the S-expression for sequence position J and the actual value of the sequence for position J. Standardized fitness equals raw fitness for this problem. The auxiliary hits measure is defined so as to count an exact match as a hit. Thus, the number of hits can range between 0 and 20.

Note that the values of this sequence range over more than five orders of magnitude

In the initial random generation of one run, the raw fitness of the worst single individual in the population was about 3 x $10^{13}$; the average raw fitness of the initial random generation was about 6 x $10^{10}$; and the raw fitness of the best single individual was 143,566.

By generation 38, the raw fitness of the best-of-generation individual had improved to 2740.

For generation 42, the raw fitness (i.e. error) of the best-of-generation individual had improved to 20. In a sequence whose largest element is 680,163, an error of only 20 is nearly perfect. This S-expression was

```
(+ (+ (- (* (* 0 1) (- (* 3 J) (+ (* 0 1) J))) 2) (*
(* (* 2 J) (+ 1 J)) (* (+ J J) (- J 2))))
(- (- (+ 2 0) (* (* 1 J) (- (- (- (+ (- (* 2 J) (+ 2
0)) (- J 3)) (- J 1)) (* (* 3 J) (+ J 1))) (- (- (+ J
J) (* (- (- (+ J (+ 0 J)) (- J 2)) (* (* 3 J) (+ J
1))) 3)) (* (- J 2) (- 2 J)))))) (* (- (+ 2 J) (* J
2)) (* (* J J) (- J 3))))).
```

When simplified, this S-expression for generation 42 is equivalent to

$$5j^4 + 4j^3 + 3j^2 + 2j \underline{\mathbf{+0}}.$$

Then, the following 100% correct individual S-expression emerged on generation 43:

```
(+ (+ (- (* (* 0 1) (- (* 3 J) (+ (* 0 1) J))) 2) (*
(* (* 2 J) (+ 1 J)) (* (+ J J) (- J 2))))
(- (- (+ 3 0) (* (* 1 J) (- (- (- (+ (- (* 2 J) (+ 2
0)) (- J 3)) (- J 1)) (* (* 3 J) (+ J 1))) (- (- (+ J
J) (* (- (- (+ J (+ 0 J)) (- J 2)) (* (* 3 J) (+ J
1))) 3)) (* (- J 2) (- 2 J)))))) (* (- (+ 2 J) (* J
2)) (* (* J J) (- J 3)))))
```

When simplified, this S-expression for generation 43 is equivalent to

$$5j^4 + 4j^3 + 3j^2 + 2j \underline{\mathbf{+1}}.$$

This is the desired mathematical expression.

Note that there is only one difference between the S-expressions in generation 42 and generation 43. The difference is that the emboldened and underlined (+ 2 0) sub-expression in generation 42 becomes (+ 3 0) in generation 43. This difference corresponds to a difference of the constant one in the simplified expressions. This difference corresponds to a numerical difference of one which, over the 20 fitness cases, accounts for the difference of 20 in raw fitness (i.e. sum or errors).

Induction of recursive sequences, such as the Fibonacci sequence and Hofstadter sequence, using the genetic programming paradigm is discussed in Koza[28]**.**

## 11. SIMPLE SYMBOLIC REGRESSION

The learning of the Boolean multiplexer function (Section 5) and the induction of sequences (Section 9) are both examples of the general problem of symbolic function identification (symbolic regression). In this section, we discuss symbolic regression as applied to real-valued functions over real-valued domains.

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

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

For example, suppose we are given a sampling of the numerical values from an unknown curve over 20 points in some domain, such as the real interval [-1.0, +1.0]. That is, we are given a sample of data in the form of 20 pairs $(x_i, y_i)$, where $x_i$ is a value of the independent variable in the interval [-1.0, +1.0] and $y_i$ is the associated value of the dependent variable. For example, these 20 pairs $(x_i, y_i)$ might include pairs such as

(-0.40, -0.2784)
(+0.25, 0.3320)
...................
(+0.50, 0.9375)

These 20 pairs ($x_i$, $y_i$) are the fitness cases that will be used to evaluate the fitness of any proposed S-expression.

The goal is to find a function, in symbolic form, that is a good fit or perfect fit to the 20 pairs of numerical data points. The solution to this problem of finding a function in symbolic form that fits a given sample of data can be viewed as a search for a mathematical expression (S-expression) from a hyperspace of possible S-expressions that can be composed from a set of available functions and arguments.

The first major step in using the genetic programming paradigm is to identify the set of terminals. In the artificial ant problem, the computer program processed information about whether food was present immediately in front of the ant in order to move the ant around the grid. In this problem, the information which the mathematical expression must process is the value of the independent variable X. Thus, the terminal set is

```
T = {X}.
```

The second major step in using the genetic programming paradigm is to identify the set of functions that will will be used to generate the mathematical expressions that attempt to fit the given finite sample of data. The function set for this problem might consist of addition (+), subtraction (-), multiplication (*), the protected division function (%) described previously, the sine function SIN, the cosine function COS, the exponential function EXP, and the protected logarithm function RLOG. The protected logarithm function RLOG returns 0 for an argument of 0 and otherwise returns the logarithm of the absolute value of the argument. Thus, the function set is

```
F = {+, -, *, %, SIN, COS, EXP, RLOG}
```

taking 2, 2, 2, 2, 1, 1, 1, and 1 arguments, respectively.

The third major step in using the genetic programming paradigm is to identify the fitness function. The raw fitness for this problem is the sum, taken over the 20 fitness cases, of the the absolute value of the difference (distance, error) between the value in the real-valued range space produced by the S-expression for a given value of the independent variable $x_i$ and the correct $y_i$ in the range space. In other words, fitness in this problem is the sum of the errors. This method of measuring fitness is the single most common method used herein.

The closer this sum is to zero, the better the computer program. Standardized fitness is, therefore, equal to raw fitness for this problem.

The genetic programming paradigm starts with the generation of 500 random S-expressions recursively composed from the available functions and terminals. Predictably, this initial population of random S-expressions includes a wide variety of highly unfit S-expressions.

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

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

The sum of the absolute values of the differences between this worst single individual and the 20 data points was as big as Avagadro's number, i.e. about $10^{23}$. That is, the raw fitness of this worst individual was about $10^{23}$.

The median individual in the initial random population was

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

This median individual is equivalent to

```
Cos [Cos (x2 + x -1)].
```

The sum of the absolute values of the differences between this median individual and the 20 data points was merely 23.67. That is, its raw fitness was 23.67. In other words, the distance between the curve for this median individual and the unknown curve (which actually is the quartic function $x^4+x^3+x^2+x$) averaged 1.2 for each of the 20 data points. Figure 1.36 shows that the curve for the median individual and the correct quartic curve are somewhat close.

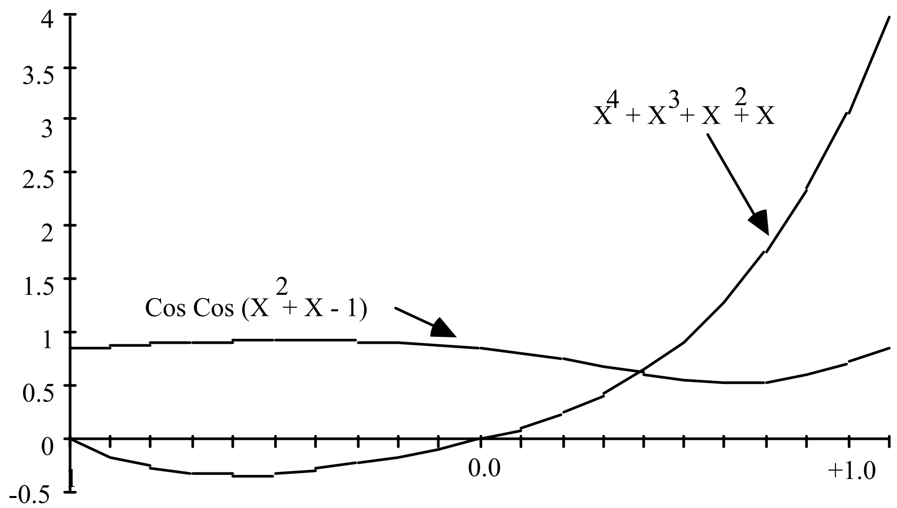


*Figure 1.36 Comparison of the median individual from generation 0 and the correct quartic curve for the symbolic regression problem*

The second best individual in the initial random population, when simplified, was

```
x + [RLog 2x + x] * [Sin 2x + Sin x²]
```

The sum of the absolute values of the differences between this second best individual and the 20 data points was 6.05. That is, its raw fitness was 6.05. The average distance between the curve for this second best individual and the unknown curve $x^4+x^3+x^2+x$ for the 20 points was about 0.3 per data point. Figure 1.37 shows that the curve for the second best individual is considerably closer to the target curve than the median individual above.

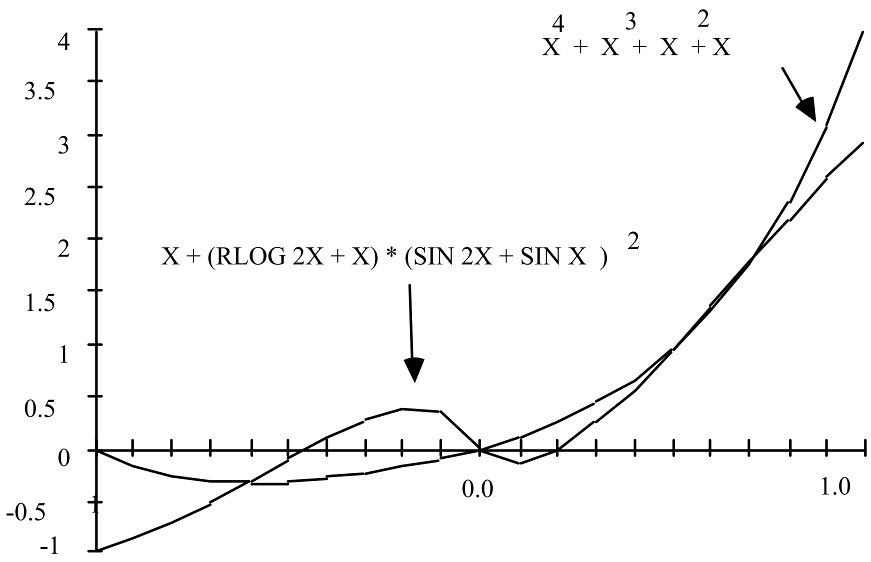


*Figure 1.37 Comparison of the second best individual from generation 0 and the correct quartic curve for the symbolic regression problem*

The best single individual in the population at generation 0 was the S-expression below with 19 points:

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

This S-expression is equivalent to $xe^x$.

The sum of the absolute value of the differences between this best-of- generation individual and the unknown curve $x^4+x^3+x^2+x$ for the 20 data points was 4.47 That is, its raw fitness was 4.47. The average distance between the curve for this best individual and the unknown curve $x^4+x^3+x^2+x$ for the 20 points is about 0.22 per data point. Figure 1.38 shows that this best-of-generation individual is considerably closer to the target curve than the second best individual above.

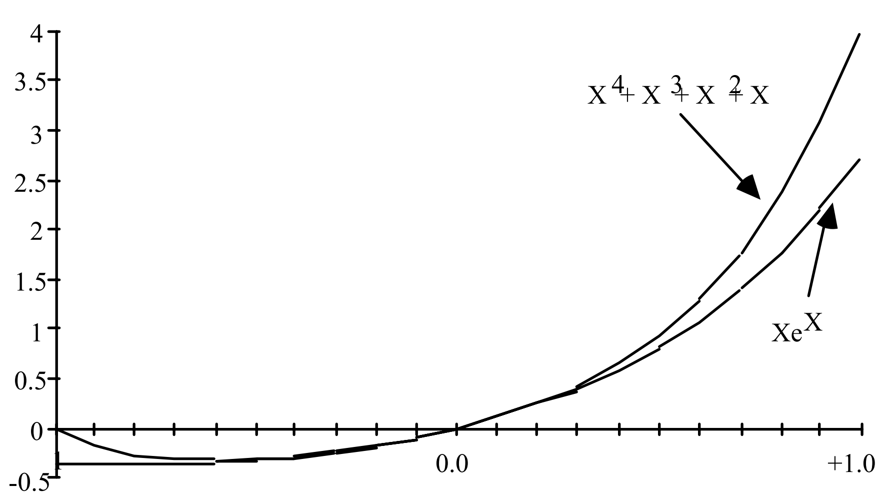


*Figure 1.38 Comparison of the best-of-generation individual from generation 0 and the correct quartic curve for the symbolic regression problem*

For this problem, we define a hit to be a fitness for which the absolute error between the S-expression and the correct curve is $\leq$ 0.01. The best-of-generation individual from the initial random population (namely $xe^x$) came within this hits criterion for 2 of the 20 fitness cases. That is, it scored 2 hits. All the other individuals in the population scored only one or zero hits.

Although $xe^x$ is not a particularly good fit (much less a perfect fit) to the unknown curve $x^4+x^3+x^2+x$, this individual is nonetheless better than the worst individual in the initial random population. It is better than the median individual. And, it is better than the second best individual. When graphed, $xe^x$ bears some similarity to the unknown target curve $x^4+x^3+x^2+x$. First, both $xe^x$ and $x^4+x^3+x^2+x$ are zero when x is zero. The exact agreement of the two curves at the origin accounts for one of the two hits scored by $xe^x$ and the closeness of the two curves for another value of x near zero accounts for the second hit. Secondly, when x approaches +1.0, $xe^x$ approaches 2.7, while $x^4+x^3+x^2+x$ approaches the somewhat nearby value of 4.0. Also, when x is between 0.0 and about -0.7, $xe^x$ and $x^4+x^3+x^2+x$ are very close.

As usual, the Darwinian reproduction operation and the genetic crossover operation are then applied to parents selected from the current population with probabilities proportionate to fitness to breed a new population of offspring computer programs. Although the vast majority of the new offspring computer programs are again highly unfit, some of them tend to be somewhat more fit than others. Moreover, over a period of time and many generations, some of them tend to be slightly more fit than those existing in earlier generations.

By generation 2, the best single individual in the population was the S-expression below with 23 points:

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

This best-of-generation individual from generation 2 is equivalent to

$$x^4 + 1.5x^3 + 0.5x^2 + x.$$

The sum of the absolute value of the differences between this best individual from generation 2 and the unknown curve $x^4+x^3+x^2+x$ for the 20 data points was 2.57 That is, the raw fitness of this best-of-generation individual improved to 2.57 for generation 2 as compared to 4.47 from generation 0. This is an average of about 0.13 per data point. This best-of-generation individual from generation 2 scored 5 hits as compared to only 2 hits for the best-of-generation individual from generation 0.

This best-of-generation individual from generation 2 bears much greater similarity to the target function than any of the predecessors discussed above. It is, for example, a polynomial. Moreover, it is a polynomial of the correct order (i.e. 4). Moreover, the coefficients of two of the four terms of this polynomial are correct (namely the coefficient of the quartic term $x^4$ and the coefficient of the linear term x). In addition, the incorrect coefficients (1.5 for the cubic term and 0.5 for the quadratic term) are not too different from the correct coefficients (1.0 and 1.0).

Notice that even though no numerical coefficients were explicitly provided in the terminal set, the rational coefficient 0.5 for the quadratic term $x^2$ was created by the process by first creating $\frac{1}{2X}$ (by dividing $\frac{X}{X} = 1$ by $X+X = 2X$) and then multiplying by X.

Similarly, the fractional rational coefficient 1.5 for the cubic term $x^3$ was created.

By generation 34, the sum of the absolute values of the differences between the best single individual and the unknown curve $x^4+x^3+x^2+x$ for the 20 data points was 0.0. That is, the raw fitness of the best single individual in the population for generation 34 attained the perfect value of 0.0. This individual, of course, also scored 20 hits.

This best-of-generation individual for generation 34 was the S-expression

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

Note that the cosine term (COS (- X X)) evaluates merely to 1.0. This entire S-expression is equivalent to $x^4+x^3+x^2+x$, which is, of course, the unknown curve.

The best-of-generation individual from generation 34 is graphically depicted in the Figure 1.39.

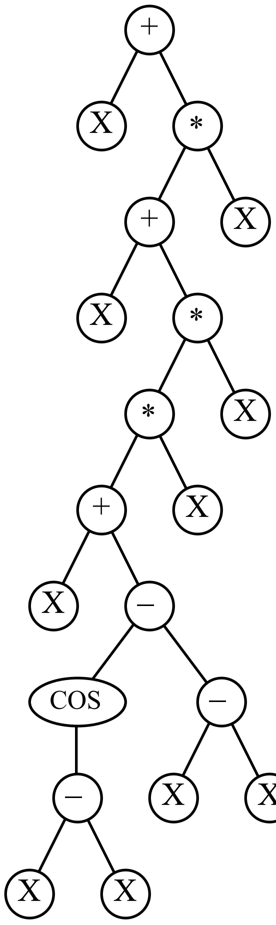


*Figure 1.39 Solution from generation 34 of the symbolic regression problem*

Note that the best-of-generation individual from generation 0 is not only not the correct solution, but it is not even of the correct functional form. Nonetheless, the genetic programming paradigm did not get trapped in this local optima (nor any subsequent local optima). Instead, the genetic programming paradigm was able to break out from sub-optimal areas of the search space and discover the correct solution to the problem.

The best-of-generation S-expression from generation 34 has 20 points. Note that there were varying numbers of points in the best-of-generation S-expression from the various intermediate generations. We did not specify that the solution would have 20 points nor did we specify the shape or content of the S-expression. The size, shape, and content of the S-expression that solves this problem evolved in response to the selective pressure provided by the fitness (error) measure.

In summary, we have shown how to use the genetic programming paradigm to find a quartic function, in symbolic form, that perfectly fits this given finite sample of data. This was achieved in spite of the fact that the function set contained numerous extraneous functions (e.g. -, %, RLOG, EXP, SIN, and COS).

## 12. SYMBOLIC REGRESSION WITH CONSTANT CREATION

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

In the previous example of symbolic regression where the unknown curve was $x^4+x^3+x^2+x$, the terminal set T consisted only of the independent variable X. There was no explicit facility for creating a numerical constant. Nonetheless, the constant 1.0 was created indirectly on two occasions via the expressions (% X X) and (COS (- X X)) and constants such as 0.5, 1.5, and other small rational constants were created with similar expressions. However, the process of symbolic regression requires a general method for discovering the appropriate numeric coefficients.

The problem of constant creation can be solved by expanding the terminal set by one terminal (called the ephemeral random constant ←). Thus, the terminal set for a symbolic regression problem with one independent variable would become

```
    T = {X, ←}.
```

During the creation of the initial random population (i.e. generation 0), whenever the ephemeral random constant ← is chosen for any point of the tree, a random number of a specified type in a specified range is generated and attached to the tree at that point.

For example, in the real-valued symbolic regression problem at hand, the ephemeral random constants are of floating point type and their range is between -1.0 and +1.0. In a problem involving integers (e.g. induction of a sequence of integers), random integers over a specified range (such as -5 to +5) are created for the ephemeral random constants "←".

Note that this random generation is done anew each time when an ephemeral "←" terminal is encountered so that the initial random population contains a variety of different random constants of the specified type. Once generated and inserted into the S-expressions of the initial random population, these constants remain fixed thereafter.

After the initial random generation, the numerous different random constants arising from the ephemeral "←" terminals will then be moved around from tree to tree by the crossover operation. These random constants will become embedded in various sub-trees that then carry out various operations on them.

This "moving around" of the random constants is not at all haphazard, but, instead, is driven by the overall goal of achieving ever better levels of fitness. For example, a symbolic expression that is a reasonably good fit to a target function may become a better fit if a particular constant is, for example, decreased slightly. A slight decrease can be achieved in several different ways. For example, there may be a multiplication by 0.90, a division by 1.10, a subtraction of 0.08, or an addition of -0.004. If a decrease of precisely 0.09 in a particular constant would produce a perfect fit, a decrease of 0.07 is usually more fit than a decrease of only 0.05. Thus, the relentless pressure of the fitness function in the natural selection process determines both the direction and magnitude of the adjustments in numerical constants

Constant creation is illustrated in the next section.

## 13. EMPIRICAL DISCOVERY

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

The problem of discovering empirical relationships from actual observed data is illustrated by the well-known non-linear econometric exchange equation

$$P=\frac{MV}{Q}.$$

This equation states the relationship between the gross national product Q of an economy, the price level P, the money supply M, and the velocity of money V.

Suppose that our goal is to find the econometric model expressing the relationship between quarterly values of the price level P and the quarterly values of the three other quantities appearing in the equation. That is, our goal is to rediscover that $P=\frac{MV}{Q}$ from the actual observed noisy time series data. Many economists believe that inflation (which is the change in the price level) can be controlled by the central bank via adjustments in the money supply M.

In particular, suppose we are given the 120 actual quarterly values (from 1959:1 to 1988:4) of following four econometric time series:

• The annual rate for the United States Gross National Product in billions of 1982 dollars (conventionally called GNP82).

• The Gross National Product Deflator (normalized to 1.0) for 1982 (called GD).

• The monthly values of the seasonally adjusted money stock M2 in billions of dollars, averaged for each quarter (called M2).

• The monthly interest rate yields of 3-month Treasury bills, averaged for each quarter (called FYGM3).

The four time series used here were obtained from the CITIBASE data base of machine-readable econometric time series (Citibank[29]).

The actual long-term historic postwar value of the M2 velocity of money in the United States is 1.6527 (Hallman et. al.[30]). Thus, the correct exchange equation for the United States in the postwar period is the multiplicative (non-linear) relationship

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

The sum of the squared errors between the actual gross national product deflator GD from 1959:1 to 1988:4 and the fitted GD series calculated from the above model over the entire 30-year period involving 120 quarters (1959:1 to 1988:4) was 0.077193. The correlation $R^2$ was 0.993320.

**MODEL DERIVED FROM FIRST TWO-THIRDS OF DATA**

We first divide the 30-year, 120-quarter period into a 20-year, 80-quarter in-sample period running from 1959:1 to 1978:4 and a 10-year, 40-quarter out-of-sample period running from 1979:1 to 1988:4. This allows us to use the first two-thirds of the data to create the model and to then use the last third of the data to test the model.

The first major step in using the genetic programming paradigm is to identify the set of terminals. The terminal set for this problem is

```
    T = {GNP82, FM2, FYGM3, ←}.
```

The terminals GNP82, FM2, and FYGM3 correspond to the independent variables of the model and provide access to the values of the time series. The ← is the ephemeral random constant terminal allowing various random floating point constants to be inserted at random amongst the initial random LISP S-expressions. In effect, the terminals for this problem are functions of the unstated, implicit time variable which ranges over the various quarters.

The second major step in using the genetic programming paradigm is to identify a set of functions. The set of functions chosen for this problem is

```
    F = {+, -, *, %, EXP, RLOG}
```

taking 2, 2, 2, 2, 1, and 1 arguments, respectively.

Notice that we are not told *a priori* whether the unknown functional relationship between the given observed data (the three independent variables) and the target function (the dependent variable, GD) is linear, multiplicative, polynomial, exponential, logarithmic, or otherwise. The unknown functional relationship could be any combination of these types of functions. Notice also that we are also not given the known constant value V for the velocity of money.

We are not told that the addition, subtraction, exponential, and logarithm function contained in the function set and the 3-month Treasury bill yields (FYGM3) contained in the terminal set are all irrelevant to finding the econometric model for the dependent variable GD of this problem.

The third major step in using the genetic programming paradigm is identification of the fitness function for evaluating how good a given computer program is at solving the problem at hand.

The fitness of an S-expression is the sum, taken over the 80 in-sample quarters, of squares of differences between the value of the price level produced by S-expression and the target value of the price level given by the GD time series.

The initial random population (generation 0) was, predictably, highly unfit. In one run, the sum of squared errors between the single best S-expression in the population and the actual GD time series was 1.55. The correlation $R^2$ was 0.49.

As before, after the initial random population was created, each successive new generation in the population was created by applying the operations of fitness proportionate reproduction and genetic recombination (crossover).

In generation 1, the sum of the squared errors for the new best single individual in the population improved to 0.50.

In generation 3, the sum of the squared errors for the new best single individual in the population improved to 0.05. This is approximately a 31-to-1 improvement over the initial random generation. The value of $R^2$ improved to 0.98. In addition, by generation 3, the best single individual in the population came within 1% of the actual GD time series for 44 of the 80 in-sample points.

In generation 6, the sum of the squared errors for the new best single individual in the population improved to 0.027. This is approximately a 2-to-1 improvement over generation 3. The value of $R^2$ improved to 0.99.

In generation 7, the sum of the squared errors for the new best single individual in the population improved to 0.013. This is approximately a 2-to-1 improvement over generation 6.

In generation 15, the sum of the squared errors for the new best single individual in the population improved to 0.011. This is an additional improvement over generation 7 and represents approximately a 141-to-1 improvement over generation 0. The correlation $R^2$ was 0.99.

In one run, the best single individual had a sum of squared errors of 0.009272 over the in-sample period. Figure 1.40 graphically depicts the best-of-generation individual.

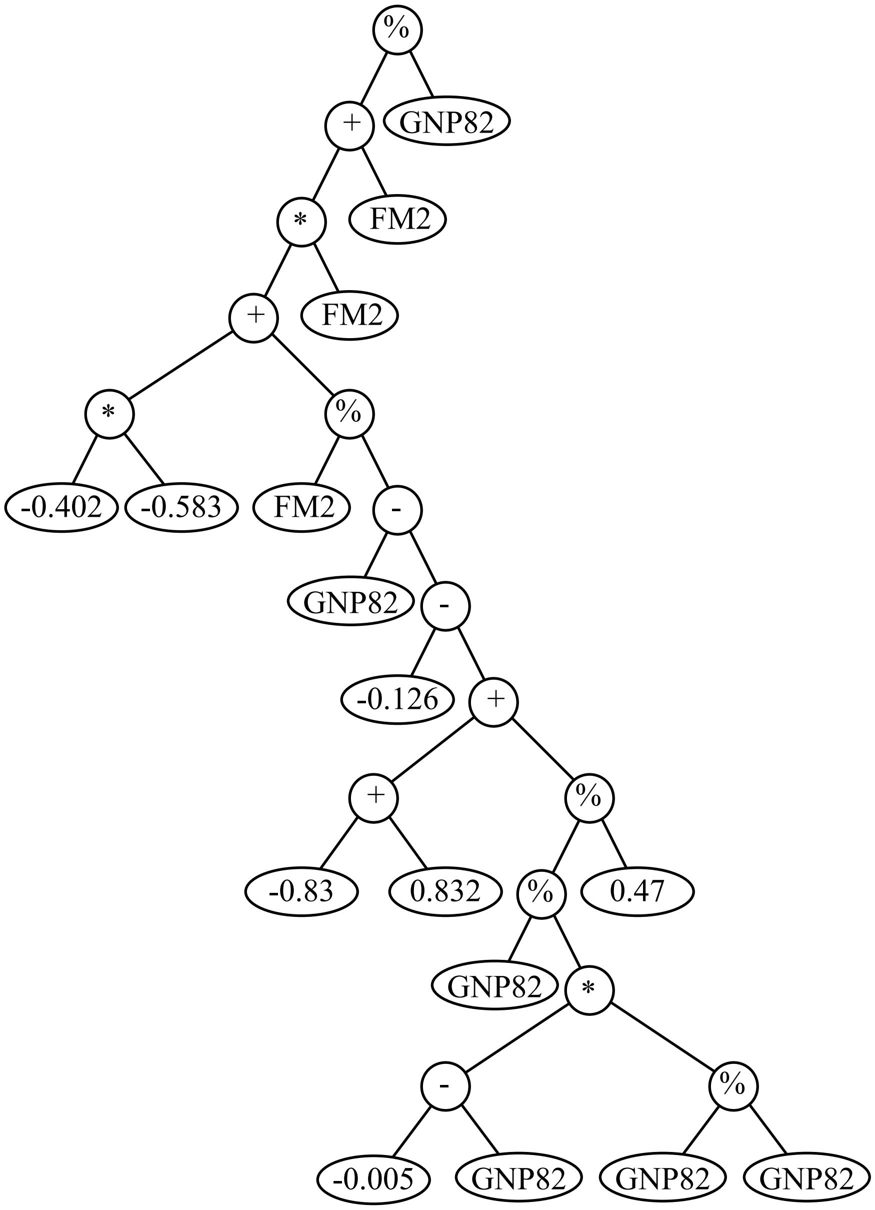


*Figure 1.40 Best result for exchange equation*

This individual is equivalent to

$$GD = \frac{(1.634 * M2)}{GNP82}$$

The table below shows the sum of the squared errors and $R^2$ for the entire 120-quarter period, the 80-quarter in-sample period, and the 40-quarter out-of-sample period:

| Data Range | 1- 120 | 1 - 80 | 81 - 120 |
|---|---|---|---|
| $R^2$ | 0.993480 | 0.997949 | 0.990614 |
| Sum of Squared Error | 0.075388 | 0.009272 | 0.066116 |

Figure 1.41 shows both the gross national product deflator GD from 1959:1 to 1988:4 and the fitted GD series calculated from the above genetically produced model for 1959:1 to 1988:4. The actual GD series is shown as a line with dotted points. The fitted GD series calculated from the above model is a simple line.

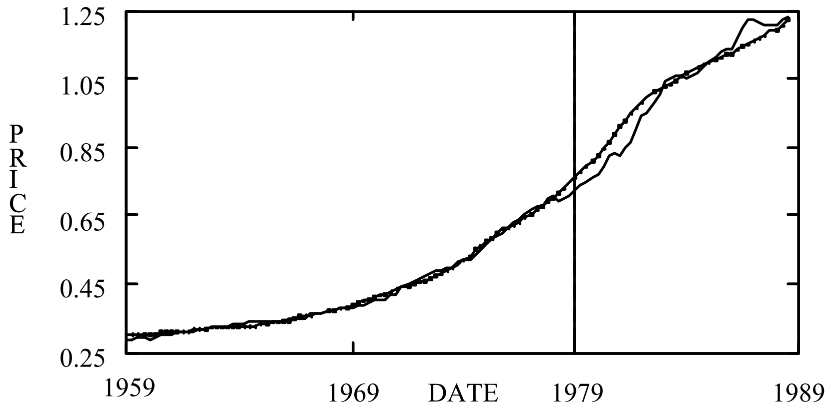


*Figure 1.41 Gross national product deflator and fitted series computed from genetically produced model*

Figure 1.42 shows the residuals from the fitted GD series calculated from the above genetically produced model for 1959:1 to 1988:4.

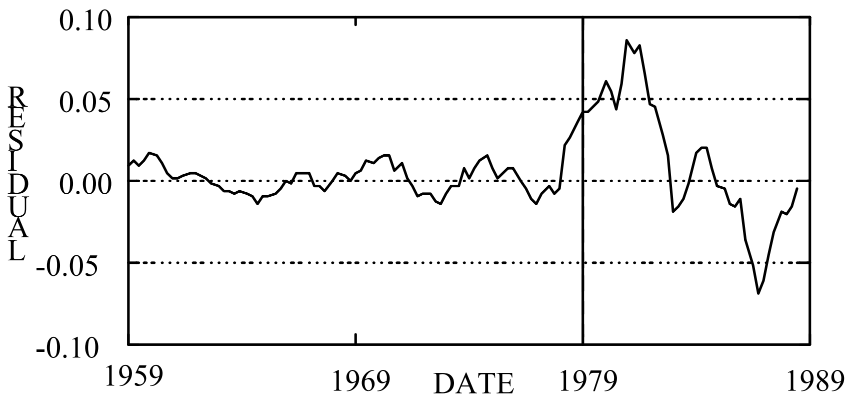


*Figure 1.42 Residuals between the gross national product deflator and fitted series computed from genetically produced model*

In Koza[31], we divide the 30-year, 120-quarter period into a 10-year, 40-quarter out-of-sample period running from 1959:1 to 1968:4 and a 20-year, 80-quarter in-sample period running from 1969:1 to 1988:4 and obtain a virtually identical model.

## 14. SYMBOLIC INTEGRATION AND DIFFERENTIATION

Symbolic integration (and differentiation) involve finding the mathematical expression which is the integral (or derivative), in symbolic form, of a given curve. Symbolic integration and differentiation are direct extensions of the symbolic regression process described in the previous section.

Symbolic integration involves finding the mathematical expression which is the integral, in symbolic form, of a given curve. The given curve may be presented either as

- a mathematical expression in symbolic form or
- a discrete sampling of data points (i.e. the symbolic form of the given curve is not explicitly specified).

If the given curve is presented as a mathematical expression, we first convert it into a finite sample of data points. We do this by taking a random sample of values $\{x_i\}$ of the independent variable appearing in the given mathematical expression over some appropriate domain. We then pair each value of the independent variable $x_i$ with the result $y_i$ of evaluating the given mathematical expression for that value of the independent variable.

Thus, regardless of the form in which the given curve is presented, we can begin the process of symbolic integration with a given finite sampling of pairs of numerical values $(x_i, y_i)$. If there are, say, 50 $(x_i, y_i)$ pairs (for i between 0 and 49), then, for convenience, we assume that the values of $x_i$ have been sorted so that $x_i < x_{i+1}$ for i between 0 and 48. The domain values $x_i$ lie in some appropriate interval.

The goal is to find, in symbolic form, a mathematical expression which is a perfect fit (or good fit) to the *integral* of the given curve using only the given 50 pairs of numerical points.

For example, if the given curve happens to be

$$\text{Cos } x + 2x + 1,$$

the goal would be to find its integral, in symbolic form, namely,

$$\text{Sin } x + x^2 + x$$

given the 50 pairs $(x_i, y_i)$. The domain appropriate to this example might be the interval $[0, 2\pi]$.

Symbolic integration is, in fact, merely symbolic regression with an additional preliminary numerical integration step. Specifically, we numerically integrate the curve defined by the given set of 50 points $(x_i, y_i)$ over the interval starting at $x_0$ and running to $x_{49}$. The integral $\mathbf{I}(x_i)$ is a function of $x_i$. The value of this integral $\mathbf{I}(x_0)$ for the first point $x_0$ is zero. For any other point $x_i$, where i is between 1 and 49, we perform a numerical integration by adding up the areas of the i trapezoids lying between the point $x_0$ and the point $x_i$. We thereby obtain an approximation to the value for the integral $\mathbf{I}(x_i)$ of the given curve for each point $x_i$. We therefore obtain 50 new pairs $(x_i, \mathbf{I}(x_i))$ for i between 0 and 49. These 50 pairs are the fitness cases for this problem.

We then perform symbolic regression in the same manner as described in Section 11 to find the mathematical expression for the curve defined by the 50 new pairs $(x_i, \mathbf{I}(x_i))$. This mathematical expression is the integral, in symbolic form, of the curve defined by the original 50 given points $(x_i, y_i)$.

In applying the genetic programming paradigm to this problem, we first define the terminal set to be

```
T = {X}.
```

Secondly, we define the function set for this problem to be

```
F = {+, -, *, %, SIN, COS, EXP, RLOG}
```

taking 2, 2, 2, 2, 1, 1, 1, and 1 arguments, respectively.

As each individual genetically produced function $f_j$ is generated, we evaluate $f_j(x_i)$ so as to obtain 50 pairs $(x_i, f_j(x_i))$. The raw fitness of an individual genetically produced function is the sum of the absolute value of difference between the value $f_j(x_i)$ of the individual genetically produced function $f_j$ at domain point $x_i$ and the value of the numerical integral **I**($x_i$). Standardized fitness equals raw fitness for this problem. A hit for this problem occurs when $f_j(x_i)$ comes within 1% of the target value **I**($x_i$).

In one run, the best single S-expression in generation 4 was the following:

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

This S-expression scored 50 hits and had standardized fitness of virtually zero. The standardized fitness (error) does not reach zero exactly due to the fact that the integral is merely a numerical approximation and because of the small errors inherent in floating point calculations.

This S-expression is equivalent to

$$\text{Sin } x + x^2 + x.$$

which is the symbolic integral of

$$\text{Cos } x + 2x + 1.$$

One could, of course, add a constant of integration, if desired.

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

Symbolic differentiation involves finding the mathematical expression which is the derivative, in symbolic form, of a given curve. The approach is similar to that of symbolic integration except that numerical differentiation is involved.

In symbolic differentiation, it is desirable to have a larger number of points for numerical differentiation (e.g. 200 points) than for numerical integration (e.g. 50 points) because of the relative inaccuracy of numerical differentiation as compared to numerical integration. Specifically, we numerically differentiate the curve defined by the given set of 200 points ($x_i$, $y_i$) over the interval starting at $x_0$ and running to $x_{199}$. The derivative **D**($x_i$) is a function of $x_i$. For any point $x_i$ other than the endpoints $x_0$ and $x_{199}$, we take the derivative to be the average of the slope of the curve between point $x_{i-1}$ and $x_i$ and the slope of the curve between point $x_i$ and $x_{i+1}$. For the two endpoints $x_0$ and $x_{199}$ of the domain, the derivative is the unaveraged slope of the curve. We thereby obtain a value for the derivative **D**($x_i$) of the given curve for each point $x_i$. We therefore obtain 200 new pairs ($x_i$, **D**($x_i$)) for i between 0 and 199. These 200 pairs are the fitness cases for this problem.

## 15. SOLVING EQUATIONS

The genetic programming paradigm can be used to solve equations where the solution comes in the form of a function that satisfies the given equation. In particular, the genetic programming paradigm can be used to solve differential equations (with given initial conditions), integral equations, general functional equations, and inverse problems.

Without loss of generality, we will assume that all equations have been transformed so that the right hand side is zero.

### 15.1. DIFFERENTIAL EQUATIONS

For some differential equations, it is possible, using exact analytic methods, to find the exact function which solves the equations. However, for most differential equations, only numerical approximation methods are available.

The problem of solving a differential equation may be viewed as the search in a hyperspace of compositions of functions and arguments for a particular composition (i.e. LISP S-expression, computer program) which satisfies the equation and its initial conditions.

**EXAMPLE 1**

Consider the simple differential equation

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

having an initial value of $\hat{y}$ of 1.0 for an initial value of $\hat{x}$ of 0.0.

The goal is to find a function which satisfies this equation and its initial condition, namely, the function $e^{-\text{Sin } x}$.

We start by generating 200 random values of the independent variable $x_i$ over some appropriate domain, such as the unit interval [0, 1]. We sort the 200 $x_i$ into ascending order.

We are seeking a function f(x) such that, for every one of the 200 values $x_i$ of the variable x, we get zero when we perform the following computation: The computation is to add the derivative $f'(x_i)$ at the point $x_i$ (i.e., $\frac{dy}{dx}$) to the product of $f(x_i)$ at point $x_i$ (i.e., y) and the cosine of $x_i$. This rewording of the problem immediately suggests an orderly general procedure for genetically finding the function f(x) that satisfies the given differential equation.

Given the set of 200 ascending values of $x_i$, we define a "curve resulting from applying the function g" to be the 200 pairs ($x_i$, $g(x_i)$), where g is some operation.

When the j-th individual genetically produced function $f_j$ in the population (i.e. S-expression) is generated by the genetic programming paradigm, we apply this function (i.e. S-expression) $f_j$ to generate a curve. Specifically, we obtain 200 values of $f_j(x_i)$ corresponding to the 200 values of $x_i$. We call these 200 pairs ($x_i$, $f_j(x_i)$) the "curve resulting from applying the genetically produced function $f_i$."

We then numerically differentiate this curve ($x_i$, $f_j(x_i)$) with respect to the independent variable $x_i$. That is, we apply the function of differentiation to obtain a new curve. Specifically, we obtain a new set of 200 pairs ($x_i$, $f_j'(x_i)$) which we can call the "curve resulting from applying the differentiation function" or "the derivative curve".

We then apply the cosine function to obtain yet another curve. Specifically, we take the cosine of the 200 random values of $x_i$ to obtain a new set of 200 pairs ($x_i$,Cos $x_i$) which we may call the "curve resulting from applying the cosine function" or "the cosine curve."

We then apply the multiplication function to the cosine curve and the y curve to obtain still another curve. In particular, we multiply the curve consisting of the set of 200 pairs ($x_i$, Cos $x_i$) by $f_j(x_i)$ so as to obtain a new curve consisting of the set of 200 pairs ($x_i$, $f_j(x_i)$*Cos $x_i$).

We then apply the addition function to this new curve and the derivative curve to obtain a new curve consisting of the set of 200 pairs ($x_i$, $f_j'(x_i) + f_j(x_i)$*Cos $x_i$).

To the extent that all 200 values of $f_j'(x_i) + f_j(x_i)$*Cos $x_i$ are close to the right hand side of the given differential equation (i.e. the zero curve) for the 200 values of $x_i$, the genetically produced function $f_j$ is a good approximation to the solution of the given differential equation. Equivalently, to the extent that the curve consisting of the

200 pairs ($x_i$, $f_j$'($x_i$) + $f_j$($x_i$)*Cos $x_i$) is close to the "zero curve" (i.e. the curve consisting of the 200 pairs (0, 0), the genetically produced function $f_j$ is a good approximation to the solution to the given differential equation.

Note that the problem of solving the given differential equation is now equivalent to a symbolic regression problem over the set of points ($x_i$, $f_j$'($x_i$) + $f_j$($x_i$)*Cos $x_i$ ).

In solving differential equations, the fitness of a particular individual genetically produced function is expressed in terms of two components. The first component is how well the function satisfies the differential equation as just described above. The second component is how well the function satisfies the initial condition of the differential equation. The first component should receive the majority of the weight in calculating fitness. We assign it 75% of the weight in the examples below. The second component receives the remainder of the weight.

The first component in computing the raw fitness of a genetically produced function $f_j$ is the sum of the absolute values of the differences between the zero function (i.e. the right hand side of the equation) and $f_j$'($x_i$) + $f_j$($x_i$)*Cos $x_i$ for i between 0 and 199, namely

$$\sum_{i=0}^{199} f_j'(x_i) + f_j(x_i)*\text{Cos } x_i$$

The closer this sum of differences is to zero, the better.

Computation of the second component in the raw fitness of a genetically produced function $f_j$ starts with the absolute value of the difference between the value of the genetically produced function $f_j(\hat{x})$ for the particular given initial condition point $\hat{x}$ and the given value $\hat{y}$ for the initial condition. Since this difference is constant over all 200 points, we multiply this difference by 200 to obtain this second component. The closer this value is to zero, the better.

For differential equations, the raw fitness of a genetically produced function $f_j$ is 75% of the first component plus 25% of the second component. The closer this overall sum is to zero, the better. Thus, standardized fitness equals raw fitness for this problem.

We now apply the above method to solving the given differential equation.

The first major step in using the genetic programming paradigm is to identify the set of terminals. The terminal set here is

```
T = {X}.
```

The second major step in using the genetic programming paradigm is to identify a set of functions. The function set for this problem is

```
F = {+, -, *, %, SIN, COS, EXP, RLOG}
```

taking 2, 2, 2, 2, 1, 1, 1, and 1 arguments, respectively.

In one run, the best individual S-expression in the initial random population (generation 0) was, when simplified, equivalent to

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

Its raw fitness was 58.09. Only 3 of the 200 points were hits. As it happens, this individual satisfies the initial condition (i.e. y

^ = 1.0 when x

^ = 0.0). This non-zero raw fitness of 58.09 (averaging 0.29 for each of the 200 points) comes entirely from the 75% component of raw fitness representing non-satisfaction of the differential equation.

By generation 2, the best-of-generation individual in the population was, when simplified, equivalent to

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

Its raw fitness was 44.23. Only 6 of the 200 points were hits. Since this individual happens to satisfy the initial condition perfectly, this raw fitness of 44.23 (i.e. 0.221 for each of the 200 points) comes entirely from non-satisfaction of the equation.

Although this best-of-generation individual from generation 2 is not a solution to the differential equation, it is a better approximation to the solution than the best-of-generation individual from generation 0.

By generation 6, the best single individual S-expression in the population was, when simplified, equivalent to

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

The raw fitness of this best-of-generation individual is down to a mere 0.057. Moreover, 199 of the 200 points are hits. Note that we do not necessarily get exactly 200 hits because of errors associated with numerical differentiation (particularly at the endpoints of the interval). Since this individual happens to satisfy the initial condition perfectly, this raw fitness of 0.057 (i.e. approximately 0.0003 for each of the 200 points) comes entirely from non-satisfaction of the equation.

This function is, in fact, the exact solution to the differential equation and its initial conditions.

The following three illustrative abbreviated tabulations of intermediate values for the best-of-generation individuals from generation 0, 2, and 6 will further clarify the above process. In each simplified calculation, we use only five equally spaced $x_i$ points in the interval [0, 1], instead of 200 randomly generated points. These five values of $x_i$ are shown in line 1.

The first calculation applies to the best-of-generation individual from generation 0, namely

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

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | $x_i$ | 0.0 | .25 | .50 | .75 | 1.0 |
| 2 | $y = e^{1 - e^{x}}$ | 1.00 | .753 | .523 | .327 | .179 |
| 3 | Cos $x_i$ | 1.00 | .969 | .876 | .732 | .540 |
| 4 | y * Cos $x_i$ | 1.00 | .729 | .459 | .239 | .097 |
| 5 | $\frac{dy}{dx}$ | -.989 | -.955 | -.851 | -.687 | -.592 |
| 6 | $\frac{dy}{dx}$ + y * Cos x | .011 | **-.225** | **.392** | **-.447** | -.495 |

Line 2 shows the value of this best-of-generation individual from generation 0 for the five values of $x_i$. Line 3 shows the cosine of each of the five values of $x_i$. Line 4 is the product of line 2 and line 3 and equals y * Cos $x_i$ for each of the five values of $x_i$.

Line 5 shows the numerical approximation to the derivative $\frac{dy}{dx}$ for each of the five values of $x_i$. For the three $x_i$ points that are not endpoints of the interval [0, 1], this numerical approximation to the derivative is the average of the slope to the left of the point $x_i$ and the slope to the right of the point $x_i$. For the two endpoints of the interval [0, 1], the derivative is the slope to the nearest point.

Line 6 is the sum of line 4 and line 5 and is an approximation to the value of the left hand side of the differential equation for the five values of $x_i$. Recall that if the S-expression were a solution to the differential equation, line 6 would be all zero or approximately zero (to match the right hand side of the equation). Of course, this best-of-generation individual from generation 0 is not a solution to the differential equation. We did not expect the values on line 6 to be zero.

The second table applies to the best-of-generation individual from generation 2, namely

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

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | $x_i$ | 0.0 | .25 | .50 | .75 | 1.0 |
| 2 | $y = e^{1 - e^{\text{Sin } x}}$ | 1.00 | .755 | .541 | .376 | .267 |
| 3 | Cos $x_i$ | 1.00 | .969 | .878 | .732 | .540 |
| 4 | y * Cos $x_i$ | 1.00 | .732 | .474 | .275 | .144 |
| 5 | $\frac{dy}{dx}$ | -.979 | -.919 | -.758 | -.547 | -.437 |
| 6 | $\frac{dy}{dx}$ + y * Cos x | .021 | **-.187** | **-.283** | **-.271** | -.292 |

Lines 1 through 5 are calculated using this best-of-generation individual from generation 2 in the same manner as above. Again, line 6 is an approximation to the value of the left hand side of the differential equation for the five values of $x_i$. The sum of the absolute value of the three non-endpoint values of line 6 is 0.74. Their average magnitude is 0.247. If we multiply this number by 200, we get 49.4. This value of 49.5 is close to the more accurate raw fitness of 44.23 obtained above with 200 points even though we are using only five $x_i$ points here (instead of 200) and the $\Delta x$ here is 0.25 (instead of an average of only 0.005). Of course, this best-of-generation individual from generation 2 is not a solution to the differential equation. We did not expect the values on line 6 to be zero.

The third calculation applies to the best-of-generation individual from generation 6, namely

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

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | $x_i$ | 0.0 | .25 | .50 | .75 | 1.0 |
| 2 | $y = e^{-\text{Sin } x}$ | 1.0 | .781 | .619 | .506 | .431 |
| 3 | Cos $x_i$ | 1.0 | .969 | .878 | .732 | .540 |
| 4 | y * Cos $x_i$ | 1.0 | .757 | .543 | .370 | .233 |
| 5 | $\frac{dy}{dx}$ | -.877 | -.762 | -.550 | -.376 | -.299 |
| 6 | $\frac{dy}{dx}$ + y * Cos x | 0.123 | **-.005** | **-.007** | **-.006** | -.067 |

Line 6 is an approximation to the value of the left hand side of the differential equation for the five values of $x_i$. Note that the three non-endpoint values in line 6 are

**-.005, -.007, -.006.**

That is, they are each very close to zero. The appearance of these near zero numbers in line 6 indicates that we have at least a good approximation to a solution to the differential equation. As mentioned above, when we use 200 points (instead of just five), the values on line 6 approximately average a mere 0.0003 for generation 6.

In summary, we solved the given differential equation for a function which satisfied the differential equation and its initial conditions.

**EXAMPLE 2**

A second example of a differential equation is

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

with initial condition such that $\hat{y} = 4$ when $\hat{x} = 1$.

In one run, the best single individual S-expression in generation 28 was

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

This is equivalent to

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

which is the exact solution to the differential equation.

## 15.2. INTEGRAL EQUATIONS

Integral equations are equations that involve the integral of the unknown function. Integral equations can be solved by the genetic programming paradigm using the same general approach and tools as described above. Of course, at some point, we take the integral of the genetically produced function,instead of a derivative.

An example of an integral equation is

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

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

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

## 15.3. INVERSE PROBLEMS

The problem of finding the inverse function is simply a problem of symbolic regression with the values of the original independent variable interchanged with the values of the original dependent variable.

## 15.4. GENERAL FUNCTIONAL EQUATIONS

General functional equations can be solved by the genetic programming paradigm using the same general approach and same tool kit as for differential equations.

## 16. PLANNING — BLOCK STACKING

Planning in artificial intelligence and robotics requires finding a plan that receives information from sensors about the state of the various objects in a system and then uses that information to select a sequence of actions to change the state of the objects in that system. We have previously seen planning in the artificial ant problem. The planning problem in this section involves an explicit iterative operation.

The block stacking problem is a robotic planning problem involving rearranging uniquely labeled blocks into a specified order on a single target tower. In the version of the problem involving nine blocks, the blocks are labeled with the nine different letters of “FRUITCAKE” or “UNIVERSAL.” The goal is to automatically generate a plan (Genesereth and Nilsson[32]) that solves this problem. This problem illustrates the use of an iterative operator DU ("Do Until") in the solution of the problem.

The STACK is the ordered set of blocks that are currently in the target tower (where the order is important). The TABLE is the set of blocks that are currently not in the target tower (where the order is not important). The initial configuration consists of certain blocks in the STACK and the remaining blocks on the TABLE (see Figure 1.43). The desired final configuration consists of all the blocks being in the STACK in the desired order (i.e. "UNIVERSAL") and no blocks remaining on the TABLE.

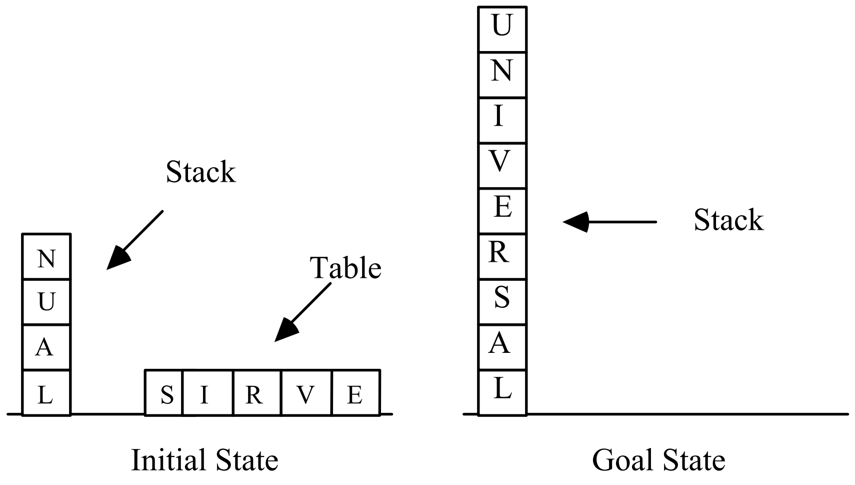


*Figure 1.43 A possible Initial State and the Goal State for the Block Stacking Problem*

Three sensors dynamically track the system. The sensor CS dynamically specifies the top block of the STACK. The sensor TB ("Top correct Block") dynamically specifies the top block on the STACK such that it and all blocks below it are in the correct order. The sensor NN ("Next Needed") dynamically specifies the block immediately after TB ("Top Correct Block") in the goal "UNIVERSAL" (regardless of whether or not there are incorrect blocks in the STACK).

Figure 1.44 shows the STACK consisting of URSAL, the sensor CS is U, while the sensor TB (the top correct block) is R since RSAL are in the correct order. The sensor NN (next needed) is E since E is the block that belongs on top of RSAL.

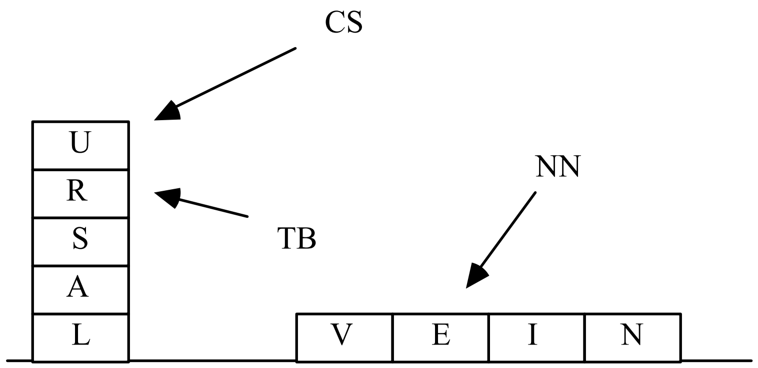


*Figure 1.44 The Initial Values for the Sensor Variables NN, TB and CS.*

The first major step in using the genetic programming paradigm is to identify the set of terminals. The terminal set T for this block stacking problem consists of the three sensors, namely,

```
    T = {TB, NN, CS}.
```

Each of these terminals is a variable atom that may assume, as its value, one of the nine block labels or NIL.

The second major step in using the genetic programming paradigm is to identify a set of functions. The function set F contains five functions

```
    F = {MS, MT, DU, NOT, EQ}
```

having 1, 1, 2, 1, and 2 arguments, respectively.

The three functions MS, MT, and DU are described below.

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

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

The iterative operator DU ("Do Until") has two arguments. The S-expression (DU *work predicate*) iteratively does the WORK until the *predicate* becomes satisfied (i.e. becomes non-NIL). The DU operator is similar to the "REPEAT...UNTIL" loop found in many programming languages. Note that the iterative DU operator differs from the typical LISP S-expression in that the *work* and *predicate* arguments are not evaluated outside the DU operator and then passed to the DU operator when the DU operator is evaluated. Instead, these arguments are evaluated dynamically inside the DU operator on each iteration. First, the *work* is evaluated inside the DU operator. Then the *predicate* is evaluated inside the DU operator. These two separate evaluations are performed, in sequence, as if the LISP function EVAL were operating inside the DU operator. Note that in an iterative construction, the execution of the *work* will often change some variable that will then be tested by *predicate*. Indeed, that is usually the purpose of the loop. Thus, it is important to suppress premature evaluation of the *work* and *predicate* arguments of the DU operator.

The genetic programming paradigm involves executing randomly generated computer programs and genetically manipulated computer programs. As a result, individual S-expressions in this problem can contain an unsatisfiable termination predicate. Thus, it is a practical necessity (when working on a serial computer) to place a limit on the number of iterations allowed by any one execution of a DU operator. Moreover, since the individual S-expressions in the genetic population often contain complicated and deep nestings of numerous DU operators, a similar limit must be placed on the total number of iterations allowed for all DU functions that may be evaluated in the process of evaluating any one individual S-expression for any one case. In particular, the DU operator times out if there have been more than 25 iterations for an evaluation of a single DU operator or if there have been a total of more than 100 iterations for all DU operators that are evaluated for a particular individual S-expression for a particular fitness case. Of course, if we could execute all the individual LISP S-expressions in parallel (as nature does) so that the infeasibility of one individual in the population does not bring the entire process to a halt, we would not need these limits.

Note that even when a DU operator times out, it nevertheless returns a value. This explicit return value resulting from the evaluation of the DU operator is, of course, in addition to the side effects that may have already been performed by the arguments to the

DU function on the state variables of the system. The return value is a Boolean value that indicates whether the *predicate* was successfully satisfied or whether the DU operator timed out.

If the predicate of a DU operator is satisfied when the operator is first called, then the DU operator does no work at all and simply returns T.

Note that the fact that each function returns some value under all conditions (in addition to whatever side effects it has on the STACK and TABLE). This guarantees closure of the function set so that every possible S-expression that might be created can be executed.

The third major step in using the genetic programming paradigm is the identification of the fitness function for evaluating how good a given computer program is at solving the problem at hand. The raw fitness of a particular individual plan (i.e. computer program) for the block stacking problem is the number of initial conditions (i.e. fitness cases) for which the particular plan produces the desired final configuration of blocks after the plan is executed. For this problem, there are millions of different fitness cases of N blocks distributed between the STACK and the TABLE. Sampling of the fitness cases is required in order to solve evaluate the fitness of a plan in a reasonable amount of time.

Thus, we construct a structured sampling of fitness cases for measuring fitness. In particular, if there are N blocks, there are N+1 fitness cases in which the blocks, if any, in the initial STACK are all in the correct order and in which there are no out-of-order blocks on top of the correctly-ordered blocks in the initial STACK. There are also N-1 additional fitness cases where there is precisely one out-of-order block in the initial STACK on top of whatever number of correctly-ordered blocks, if any, happen to be in the initial STACK. There are additional fitness cases with more than one out-of-order block in the initial STACK on top of various numbers of correctly-ordered blocks in the initial STACK. In lieu of the millions of possible fitness cases, we construct a structured sampling of fitness cases for measuring fitness consisting of the following 166 fitness cases:

- the l0 cases where the 0-9 blocks in the STACK are already in correct order,
- the 8 cases where there is precisely one out-of-order block in the initial STACK on top of whatever number of correctly-ordered blocks, if any, happen to be in the initial STACK, and

- a structured random sampling of 148 additional cases with between zero and eight correctly-ordered blocks in the initial STACK and a random number (between two and eight) of out-of-order blocks on top of the correctly-ordered blocks.

Raw fitness is the number of fitness cases (out of 166) that the plan (computer program) correctly handles. A plan correctly handles a fitness case if the STACK contains nine blocks spelling “UNIVERSAL” when it is finished.

Raw fitness ranges between 0 and 166. Standardized fitness, in turn, equals 166 minus raw fitness. A standardized fitness of zero corresponds to 166 correctly handled cases.

Obviously, the construction of a sampling such as this must be done so that the process is not misled into producing solutions that correctly handle some unrepresentative subset of the entire problem but cannot correctly handle the entire problem.

## 16.1. CORRECTLY STACKING BLOCKS

The first version of the block-stacking problem we consider involves finding a plan (i.e. computer program) which can correctly stack the nine blocks onto the STACK in the desired order after starting with any of the 166 fitness cases. Each plan is executed (evaluated) once for each of the 166 cases.

The initial random population of plans contains a variety of complicated, inefficient, pointless, and counter-productive plans. One initial random plan

```
(EQ (MT CS) NN)
```

unconditionally moves the top of the STACK to the TABLE and then performs the useless Boolean comparison between the sensor value NN and the return value of the MT function.

Another initial random plan

```
(MS TB).
```

futilely attempts to move the block TB (which already is in the STACK) from the TABLE to the STACK.

Many initial random plans are so ill-formed that they perform no action at all on the STACK and the TABLE. These plans score a raw fitness of one (out of a maximum of 166) because they leave the

STACK untouched in the one fitness case consisting of an already perfectly arranged STACK.

Other initial random plans are even more unfit and even disrupt a perfectly arranged initial STACK.

Some idiosyncratic initial random plans achieve modest fitness levels, but have no utility in general. They contain particular action sequences that happen to work on a specific two, three, or four of the fitness cases. For example, the plan

```
   (EQ (MS NN) (EQ (MS NN) (MS NN)))
```

moves the next needed block (NN) from the TABLE to the STACK three times. This plan works in the four particular specific fitness cases where the initial STACK consists of six, seven, eight, or nine correct blocks and no out-of-order blocks.

In one run, an individual plan emerged in generation 5 that correctly handled 10 of the 166 fitness cases. This plan correctly handles the 10 fitness cases in the first group itemized above where the blocks, if any, initially on the STACK happen to already be in the correct order and where there are no out-of-order blocks on top of these correctly-ordered blocks. This plan was

```
    (DU (MS NN) (NOT NN)).
```

This plan uses the iterative operator DU to do the work (MS NN) of moving the needed block onto the STACK from the TABLE until the predicate (NOT NN) is satisfied. This predicate is satisfied when there are no more blocks needed to finish the STACK (i.e. the “next needed” sensor NN is NIL).

Figure 1.45 shows this partially correct plan moving five needed blocks (E, V, N, I, and U) to a STACK containing four blocks (R, S, A, and L) that are already in the correct order.

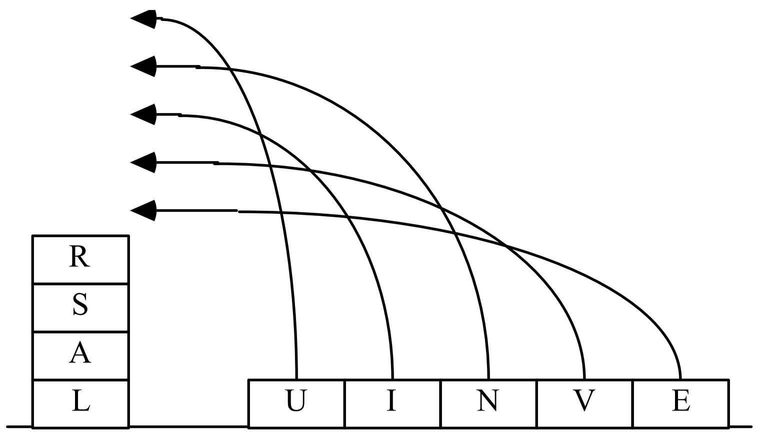


*Figure 1.45 Five blocks to go on top of RSAL being moved from TABLE to STACK*

This plan, of course, does not produce a correct final STACK if any block initially on the STACK was incorrect. Thus, this plan performs incorrectly in 156 of the 166 fitness cases. Nonetheless, this partially correct plan will prove to be be a useful “building block” in the final 100% correct plan.

As additional generations are run, the performance of the best single individual plan in the population typically increases somewhat from generation to generation. These progressively improving plans each deal correctly with a few more additional cases. At the same time, the overall average fitness of the population also tends to increase somewhat from generation to generation as the population begins to contain additional higher scoring plans.

In generation 10 of one run, the best single individual plan in the population achieved a perfect score (that is, the plan produced the desired final configuration of blocks in the STACK for 100% of the fitness cases). This 100% correct plan was

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

Figure 1.46 graphically depicts this 100% correct plan from generation 10.

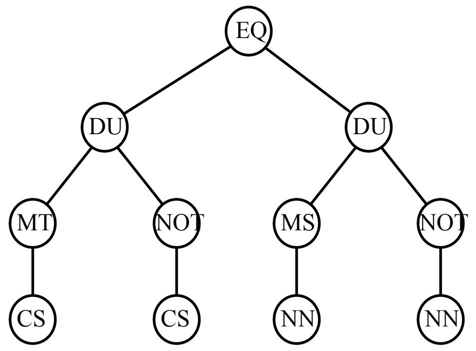


*Figure 1.46 100% correct, but inefficient, plan for stacking blocks from generation 10*

This plan consists of two sub-plans which are connected via the function EQ (which is merely serving as a connective). The first sub-plan is

```
(DU (MT CS) (NOT CS))
```

This sub-plan does the work of first moving CS (i.e. the top of the STACK) to the TABLE. This continues until the predicate `(NOT CS)` becomes T (True). This predicate becomes true when the top of the STACK becomes NIL (i.e. the STACK becomes empty).
The second sub-plan

```
(DU (MS NN) (NOT NN))
```

does the work of iteratively moving the next needed block NN to the STACK until there is no remaining next needed block NN.

Notice that the previously discovered partially correct plan

```
(DU (MS NN) (NOT NN)),
```

was incorporated as a subplan into the 100% correct final hierarchy. This subplan became part of the hierarchy as a result of the crossover operation. This subplan participated in the critical crossover operation because its relatively high fitness (i.e. a raw fitness of 10 out of a possible 166) allowed it to be selected as a parent to participate in the crossover operation which produced the 100% correct final plan.

## 16.2. EFFICIENTLY STACKING BLOCKS

The 100% correct solution found in the basic version of the block stacking problem described above is highly inefficient in that it removes all the blocks, one by one, from the STACK to the TABLE (even if they are already in the correct order on the STACK). This plan then moves the blocks, one by one, from the TABLE to the STACK. As a result, this plan uses 2319 block movements to handle the 166 cases. We should not be surprised by this since the concept of efficiency was not involved in any way in the basic version of the problem described above.

The most efficient way to solve this version of the block stacking problem, in terms of minimizing total block movements, is to remove only the out-of-order blocks from the STACK and then to move the next needed blocks to the STACK from the TABLE. This approach uses only 1641 block movements to handle the 166 fitness cases.

Note, however, that nothing in the fitness function we defined above for the basic version of the block stacking problem gave any consideration whatsoever to efficiency as measured by the total number of block movements involved. The only consideration in the fitness function that we defined above was whether or not the plan (computer program) correctly handled each of the 166 fitness cases (either at its natural time of termination or when it timed out).

We can, however, simultaneously breed a population of plans (computer programs) for two attributes at one time. In particular, we can specifically breed a population of plans for *both* correctness and efficiency by using a combined fitness measure that assigns a majority of the weight (say 75%) to correctness and a minority of weight (say 25% ) to a secondary attribute (i.e. efficiency).

Specifically, if a plan correctly handles all 166 fitness cases, it would receive 75 points towards the combined fitness measure. If a plan correctly handles zero cases out of 166, it would receive zero points towards the combined fitness measure. If a plan correctly handled 40% of the 166 cases (i.e. 67 cases), it would receive 30 points (40% of 75) towards the combined fitness measure. Then, if that plan took 1641 block movements, it would receive 25 additional points towards the combined fitness measure. If the plan took between 0 and 1640 block movements to perform its work, the 25 points available for efficiency would be scaled linearly upwards so that a plan making zero block movements would receive zero of the 25 points. If the plan made between 1642 and 2319 block movements, the 25 points available for efficiency would be scaled linearly downwards so that a plan making 2319 block movements would receive zero of the 25 points. If a plan made more than 2319 block movements, it would also receive zero of the 25 points available for efficiency.

In one run, the best single individual from the initial random population performed correctly in only l of the 166 cases and made a total of 6590 block movements. This plan was both incorrect and inefficient.

However, by generation 11, the best individual in the population was

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

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

This l00% correct and 100% efficient plan is graphically depicted in Figure 1.47.

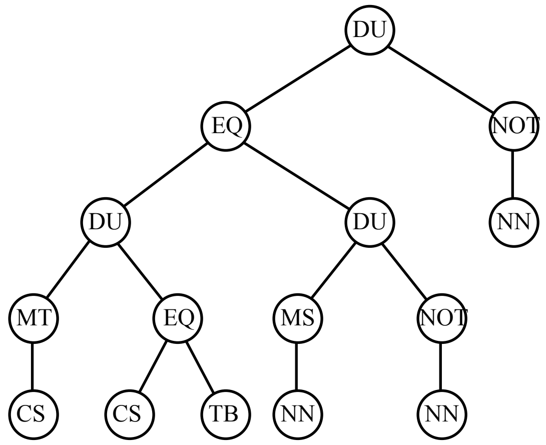


*Figure 1.47 100% correct and 100% efficient solution to the block stacking problem*

In this plan, the sub-plan

```
(DU (MT CS) (EQ CS TB))
```

iteratively moves CS (the top block) of the STACK to the TABLE until the predicate `(EQ CS TB)` becomes satisfied. This predicate becomes satisfied when CS (the top of the stack) equals TB (top correct block in the STACK).

Figure 1.48 shows the out-of-order blocks (I and V) being moved to the TABLE from the STACK until R becomes the top of the STACK. When R is the top of the STACK, CS equals TB.

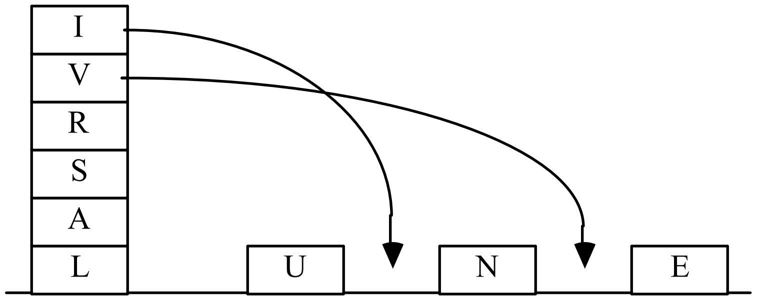


*Figure 1.48 Out-of-order blocks I and V being moved from STACK to the TABLE*

Then, the previously discovered second sub-plan

```
(DU (MS NN) (NOT NN))
```

iteratively moves the next needed blocks (NN) to the STACK until there is no longer any next needed block.

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

## 17. OPTIMAL CONTROL

Problems of optimal control involve a system that is described by state variables. The future state of the system is determined by the choice of certain control variables. The objective in optimal control is to choose the control variables so as to cause the system to go to a specified target state with an optimal (typically minimal) cost.

The problem of balancing a broom in minimal time by applying a bang-bang force from either direction is a well-known optimal control problem (Widrow[33]). The broom balancing problem involves a push cart with mass $m_c$ moving on a one dimensional frictionless track and a broom (inverted pendulum) of mass $m_b$ pivoting on the top of the cart. The broom has an angle $\theta$ (measured from the vertical) and an angular velocity $\omega$. The distance from the center of mass of the broom to the pivot is $\lambda$.

There is one control variable for this system, namely, a bang-bang force F of fixed magnitude which can be applied to the center of mass of the cart at each time step so as to accelerate the cart towards either the positive or negative direction along the track.

There are four state variables of this system, namely, position x, velocity v, angle $\theta$, and angular velocity $\omega$.

Figure 1.49 shows the cart at time t with position x(t), velocity v(t), an angle $\theta$(t) between the broom and the vertical, and angular acceleration $\omega$ (t). The bang-bang force is being applied so as to accelerate the cart towards the positive direction.

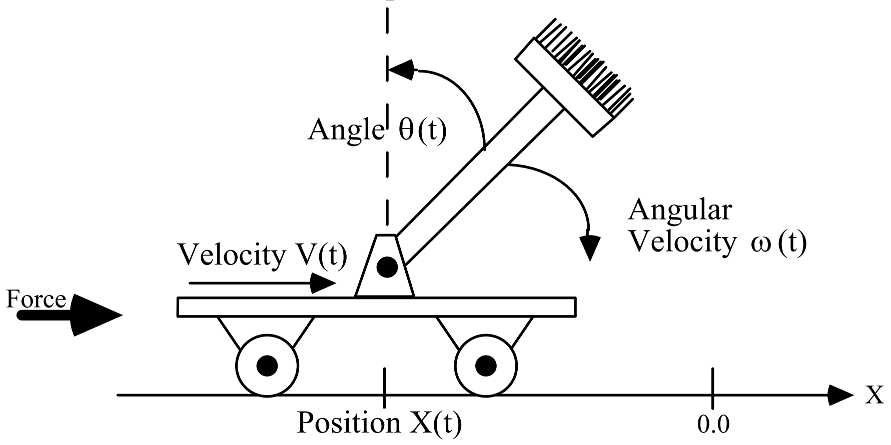


*Figure 1.49 Moving cart with pivoting broom*

At each time step, the choice of value of the control variable (i.e. the quantity u equal to a multiplier of either +1 or -1 to the magnitude $|F|$ of the force F) at time step t causes a change in the state variables of the system at time step t+1.

The state transitions of this system are expressed by non-linear differential equations. At each discrete time step $\tau$, the current state of the system and the force being applied at that time step are used to compute the state of the system at the next time step.

In particular, the angular acceleration of the broom $\Phi$ (t) at time t is given by (Anderson[34]) as

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

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

$$\omega(t+1) = \omega(t) + \tau\,\Phi(t)$$

Then, as a result of this angular acceleration $\Phi(t)$, the angle $\theta(t+1)$ at time t+1 is, using Euler approximate integration,

$$\theta(t+1) = \theta(t) + \tau\ \omega(t).$$

The acceleration a(t) of the cart on the track is given by

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

The velocity v(t+1) of the cart on the track at time t+1 is therefore

$$v(t+1) = v(t) + \tau\,a(t).$$

The position x(t+1) of the cart on the track at time t+1 is

$$x(t+1) = x(t) + \tau\,v(t).$$

The problem is to find a time-optimal control strategy (i.e. a computer program) for balancing the broom that satisfies the following three conditions:

- The control strategy (computer program) specifies how to apply the bang-bang force at each time step for any combination of the state variables .
- The system comes to rest with the broom balanced (i.e. reaches a target state with approximate speed 0.0, approximate angle $\theta$ of 0.0, and approximate angular velocity $\omega$ of 0.0).
- The time required is minimal.

The constants here are $m_c = 0.9$ kilogram, $m_b = 0.1$ kilogram, gravity g=1.0 meters/sec$^2$, time step $\tau$=0.02 seconds, and length $\lambda$ = 0.8106 meters.

The first major step in using the genetic programming paradigm is to identify the set of terminals. In this section, we consider only the particular version of the broom balancing problem that controls the three state variables of velocity v, angle $\theta$ , angular velocity $\omega$). Thus, the terminal set for this problem is

```
T = {v, θ, ω , ←},
```

where ← is the ephemeral random constant for floating point random numbers.

The second major step in using the genetic programming paradigm is to identify the set of functions. The function set for this problem consists of addition, subtraction, multiplication, division, the sign function (SIG), the absolute value function (ABS), the square root of absolute value function (SRT), the square function (SQ), the cube function (CUB), and the greater-than function (GT). The greater-than function GT is a function of two arguments which returns +1 if its first argument is greater than its second argument and returns -1 otherwise.

That is, the function set for this problem is

```
    F= {+, -,  *, %, SIG, ABS, SRT, SQ, CUB, GT}
```

taking 2, 2, 2, 2, 1, 1, 1, 1, 1, and 2 arguments, respectively.

The use of the protected division function (%), the square root of absolute value function (SRT), and the greater-than function (GT) together guarantee closure of the function set.

Since we do not know the exact mathematical solution to this problem, we have no guarantee as to the sufficiency of the function set.

We included both the SIG and ABS functions, the square function (SQ), and cube function (CUB) merely because these extraneous functions might prove useful. Each additional function in the function set of the genetic programming paradigm usually slightly reduces the efficiency of the run. On the other hand, the solution produced by the genetic programming paradigm rarely comes in the precise form we anticipate. Moreover, the large benefits produced by having one additional function in facilitating a rapid and relatively parsimonious solution often far overweights the slight cost of having one extraneous function in the function set. For example, the artificial ant problem (Section 7) can be solved without having both the LEFT and RIGHT functions; however, the solution comes much more slowly and the solution is far less parsimonious. We believe that, when in doubt, it is often better to include potentially extraneous functions.

The third major step in setting up the genetic programming paradigm is to identify the fitness function for the problem. Each control strategy is executed (evaluated) on every time step of each fitness case. This problem requires an output interface (wrapper) to guarantee that every value returned as a result of the execution of a control strategy is translated into some appropriate bang-bang force. For this problem, the wrapper specifies that any positive numerical output will be interpreted so as to apply the bang-bang force F to accelerate the system towards the positive direction. Any other output (of whatever type) will be interpreted so as to apply the bang-bang force F to accelerate the system towards the negative direction.

If an output interface (wrapper) is needed at all, the nature of the wrapper needed by a particular problem flows from the choice of the terminal set and the function set for the problem. Most problems do not require any wrapper in the genetic programming paradigm because we are free to choose the functions and terminals in terms that are very natural for the problem. This particular problem requires a wrapper since we want a binary result (i.e. the bang-bang force) whereas the state variables (terminals) and functions applied to the state variables are real-valued. If a wrapper is required at all, it is typically a very simple one (as is the case here). The randomizer problem (Section 9) required a similar wrapper to create a binary result.

Figure 1.50, shows a control surface partitioning the three dimensional $v$-$\theta$– $\omega$ state space. When the system is at a point $(v, \theta, \omega)$in the state space that is above the control surface, the force F is applied so as to accelerate the cart towards the positive direction. Otherwise, the force is applied so as to accelerate the cart towards the negative direction.

*Figure 1.50 Illustrative control surface partitioning the three dimensional state space for the broom balancing problem*

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

Time was discretized into 300 time steps of .02 seconds. The total time available before the system times out for a given control strategy is thus 6 seconds. If the square root of the sum of the squares of the velocity v, angle θ, and angular velocity ω is less than 0.07 (the hit criterion), the system is considered to have arrived at its target state (i.e. with the broom balanced and the cart at rest). If a particular control strategy brings the system to the target state for a particular initial condition case, its fitness for that initial condition case is the time required (in seconds). If a control strategy fails to bring the system to the target state before it times out for a particular initial condition case, its fitness for that case is set to 6 seconds (i.e. the maximum). The “fitness” of a control strategy is the average time for the strategy over all 10 fitness cases.

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

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

The best single control strategy in generation 0 was the non-linear control strategy

$$v^2 + \theta$$

which averaged 3.77 seconds. Note that this control strategy is partially blind in that it does not even consider the state variable $\omega$ in specifying how to apply the bang-bang force.

The average population fitness improved to 5.27, 5.23, 5.15, 5.11, 5.04, and 4.97 seconds per fitness case in generations 1 through 6, respectively.

The best single individual of generation 4 was the simple linear control strategy

```
(+ (+ ANG AVL) AVL).
```

This S-expression is equivalent to

$$\theta + 2\,\omega.$$

The control surface corresponding to this S-expression is merely a plane. In generation 6, the best single individual was the non-linear control strategy

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

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

By generation 10, the average population fitness had improved further to 4.8 seconds and scored 8 hits. The best single individual was

$$\theta + 2\,\omega - v^2.$$

Note that this individual is not partially blind and considers all three state variables.

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

The best single individual of generation 16 is the S-expression

```
(+ (* (SQ (+ ANG AVL)) (SRT AVL))
   (+ (- ANG (SQ VEL)) AVL))
```

This s-expression is equivalent to

$$\sqrt{\omega}\,(\theta+\omega)^2+\omega\;+\theta-v^2$$

Figure 1.51 shows the highly non-linear control surface corresponding to this S-expression for generation 16.

*Figure 1.51 Control surface for best-of-generation individual for generation 16 of broom balancing problem*

The best single individual of generation 20 is the S-expression

```
(+ (* (ABS (SQ (+ ANG AVL))) (SRT AVL))
      (+ (- ANG (SQ VEL)) AVL))
```

This s-expression is equivalent to

$$\sqrt{\omega}\left|(\omega\;+\theta\,)^2\right|\;+\omega\;+\;\theta\;-v^2$$

The highly non-linear control surface corresponding to this S-expression for generation 20 is only slightly different from that of generation 16.

In generation 24, we attained one individual that scored 10 hits. The best individual in generation 24 is, when simplified, the non-linear control strategy

$$v+\theta+2\;\omega\;+\theta^{\,3}.$$

This individual had a raw fitness of 2.63 seconds.

By generation 24, the population average fitness improved to 4.2 seconds.

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

$$v + 2\,\theta + 3\,\omega\,.$$

This individual scored 10 hits and had fitness of 2.16 seconds. Note that the computer program or control strategy can be viewed as defining the optimal control surface that partitions the state space into parts. For this particular generation, the control surface is merely a plane.

In generation 33, the best single individual bears a resemblance to the ultimate solution we attain in generation 46. In generation 33, the best single individual is

$$8\,\omega^3 + v + \theta + \omega\,.$$

This individual had fitness 1.57 seconds. Moreover, 15% of the individuals in the population in generation 33 scored 10 hits.

The best single individual for generation 34 was the S-expression

```
(+ (+ (+ (CUB (+ AVL AVL)) (+ VEL AVL)) ANG)
   (ABS (ABS (SQ (* (* (SRT 0.24000001)
                       (+ (SRT ANG) AVL))
                    (ABS VEL))))))
```

This s-expression is equivalent to

$$(2\omega)^3 + v + \omega + \theta + \left|\left[\sqrt{0.24}(\sqrt{\theta} + \omega)\left|v\right|\right]^2\right|$$

Figure 1.52 shows the non-linear control surface corresponding to this S-expression for generation 34.

*Figure 1.52 Control surface for the best-of-generation individual for generation 34 of the broom balancing problem*

By generation 35, the high point of the hits histogram of the population moved from 4 hits to 10 hits. In particular, 30% of the individuals in the population scored 10 hits. The best single individual for generation 35 was

```
(+ (+ (+ (CUB (+ AVL AVL)) (+ VEL AVL)) ANG)
   (ABS (ABS (SQ ANG))))
```

This s-expression is equivalent to

$$(2\omega)^3 + v + \omega + \theta + \theta^2$$

The best single individual for generation 40 was the S-expression

```
(+ (+ (+ (CUB (+ AVL AVL)) (+ VEL AVL)) ANG)
   (+ (+ (CUB (+ (+ VEL AVL) ANG)) (+ VEL AVL)) ANG))
```

This S-expression is equivalent to

$$(2\omega)^3 + 2(v + \omega + \theta) + (v + \omega + \theta)^3$$

The best single individual for generation 44 was the S-expression

```
(+ (+ ANG AVL) (+ (+ (+ (CUB (+ AVL AVL))
                        (+ VEL AVL)) ANG) VEL))
```

This s-expression is equivalent to

$$2(v + \omega \ + \theta ) + (2\omega )^3$$

Finally, in generation 46, the best single individual in the population was the S-expression

```
(+ (+ (+ (CUB (+ AVL AVL)) (+ VEL AVL)) ANG)
   (+ (* (+ VEL AVL) (+ (SRT ANG) AVL)) ANG))
```

This S-expression corresponds to the following 8-term non-linear control strategy:

$$v + 2\theta + \omega + 8\omega^3 + \omega^2 + v\omega + v|\theta + \omega|\theta .$$

Figure 1.53 shows the non-linear control surface corresponding to this S-expression for generation 46.

*Figure 1.53 Control surface for the best-of-generation individual for generation 46 of the broom balancing problem*

Figure 1.54 shows the progressive improvement (decrease) during this run of the average raw fitness of the population and the best-of-generation individual. The raw fitness of the worst single individual in the population is at the top of the graph for almost every generation indicating the presence of at least one individual that timed out for all 10 fitness cases.

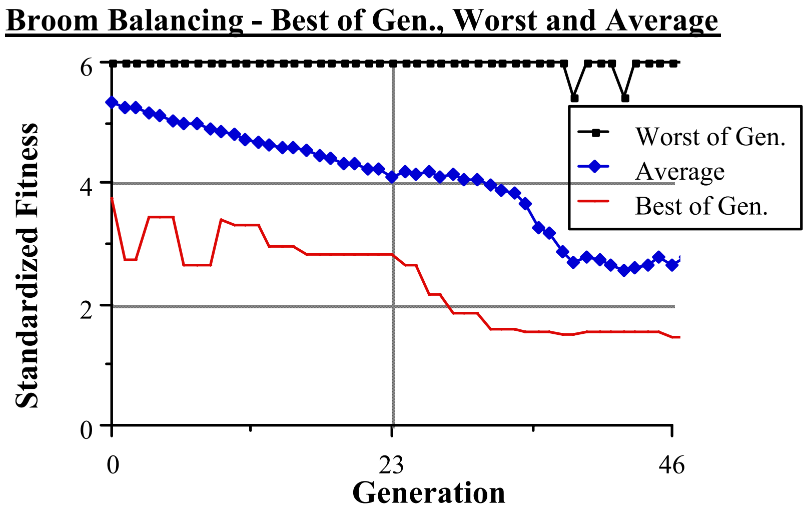


*Figure 1.54 Standardized fitness for worst-of-generation individual, average standardized fitness for population, and standardized fitness for best-of-generation individual for broom balancing problem*

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

The decision as to when to terminate a run presents some difficulty in optimization problems in general since we are seeking both the unknown optimal time and the computer program that achieves this unknown time.

After its discovery, this single best control strategy found in generation 46 was retested on 1000 additional random initial condition points. It performed in an average of 1.51 seconds.

In another test, this control strategy from generation 46 averaged 2.65 seconds when the initial conditions consisted of the 8 corners of the v- $\theta$ - $\omega$ cube. In another test, it took 4.24 seconds when the initial conditions consisted of the hardest two corners of the cube (i.e. where the velocity, angle, and angular velocity have the same sign). This control strategy never timed out for any internal point or any corner point of the cube.

A pseudo optimum strategy developed by Keane (Koza and Keane[35,36]) served as an approximate guide for verifying the attainment of the optimal value for time. This pseudo optimum strategy is an approximate solution to a linear simplification of the problem.

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

These results (in average seconds per fitness case) are summarized in the table below:

*PERFORMANCE FOR 3-DIMENSIONAL BROOM BALANCING PROBLEM*

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

We know of no control strategy for this formulation of the broom balancing problem as good as

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

from generation 46 of the run described above. We do know that this control strategy had the best time of the many similar control strategies that we discovered; that there were numerous other control strategies that were only slightly worse (thereby suggesting possible convergence); and that this particular control strategy is slightly better than the benchmark psuedo optimum strategy developed by Keane.

Figure 1.55 graphically depicts the control strategy from generation 46.

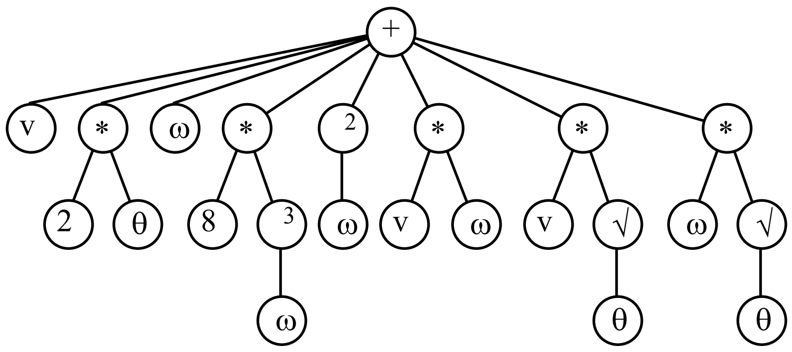


*Figure 1.55 Best-of-generation individual from generation 46 of the broom balancing problem.*

## 18. MINIMAX STRATEGY FOR A GAME

The problem of discovering a strategy for playing a game is an important problem in game theory.

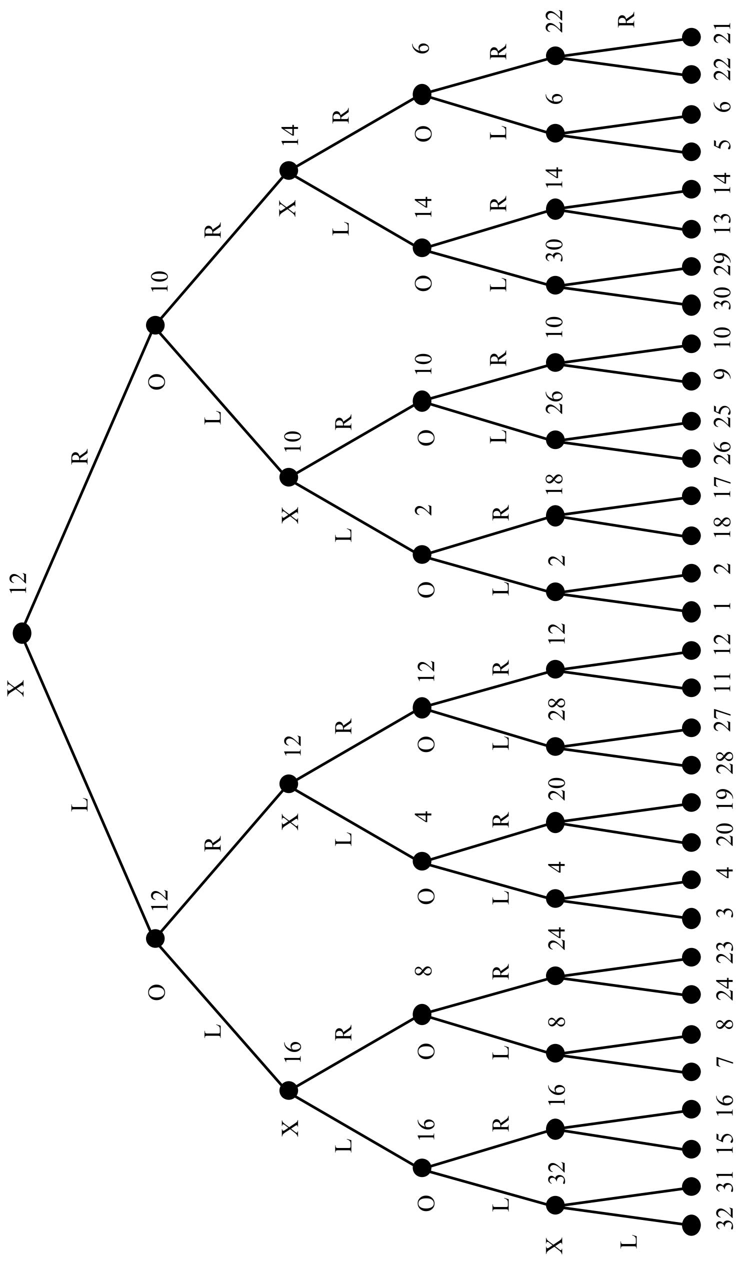
In a game, there are two or more independently-acting players who make choices (moves) and receive a payoff based on the choices they make.

A strategy for a given player in a game is a way of specifying what choice the player is to make at a particular point in the game from all the allowable choices at that time, given all the information that is available to the player at that time.

The problem of discovering a strategy for playing a game can be viewed as requiring the discovery of a computer program. Depending on the game, the desired computer program takes as its input either the entire history of past moves in the game or the current state of the game. The desired computer program then produces the next move as its output.

Consider the discrete 32-outcome game whose game tree is presented in extensive form in Figure 1.56.

*Figure 1.56  32-outcome game tree with payoffs*



This game is a two-person, competitive, zero-sum 32-outcome game in which the players make alternating moves. On each move, a player can choose to go L (left) or R (right). Each internal point of this tree is labeled with the player who must move. Each line is labeled with the choice (either L or R) made by the moving player. Each endpoint of the tree is labeled with the payoff (to player X). After player X has made three moves and player O has made two moves, player X receives (and player O pays out) the particular payoff shown at the particular endpoint of the game tree.

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

A strategy for a particular player in a game specifies which choice that player is to make for every possible situation that may arise for that player. In particular, in this game, a strategy for player X must specify his first move if he happens to be at the beginning of the game. Second, a strategy for player X must also specify his second move if player O has already made one move. Third, a strategy for player X must also specify his third move if player O has already made two moves.

Since Player X moves first, player X's first move is not conditioned on any previous move. But, player X's second move will depend on Player O's first move (i.e. OM1) and, in general, it will also depend on his own first move (XM1). Similarly, player X's third move will depend on player O's first two moves and, in general, his own first two moves.

Similarly, a strategy for player O must specify what choice player O is to make for every possible situation that may arise for player O.

A strategy here is a computer program whose inputs are the relevant historical variables (XM1, OM1, XM2, and OM2) and whose output is a move (L or R) for the player involved. Note that there is no reference to XM3 since the game ends as soon as X makes his third move.

Four testing functions CXM1, COM1, CXM2, and COM2 provide the facility to test each of the historical variables (XM1, OM1, XM2, and OM2) that are relevant to deciding upon a player's move. Each of these functions is a specialized form of the CASE function in LISP. For example, function CXM1 has three arguments and evaluates its first argument if XM1 (X's move 1) is undefined, evaluates its second argument if XM1 is L (Left), and evaluates its third argument if XM1 is R (Right). Functions CXM2, COM1, and COM2 are similarly defined.

The first and second major steps in using the genetic programming paradigm are to identify the terminal set and the function set. The terminal set for this problem is

```
T = {L, R}.
```

The function set is

```
F = {CXM1, COM1, CXM2, COM2}.
```

Each of these functions takes three arguments.

The third major step in using the genetic programming paradigm is to identify the fitness function. The fitness of a particular strategy for a particular player in a game is the average payoff received when that strategy is played against all possible sequences of combinations of moves by the opposing player.

Thus, when we compute the fitness of an X strategy, we must test the X strategy against all 4 possible combinations of O moves — that is, O's choice of L or R for his moves 1 and 2. When we compute the fitness of an O strategy, we must test it against all 8 possible combinations of X moves — that is, X's choice of L or R for his moves 1, 2, and 3. Note that it is not sufficient, in general, for player X to play only against an opponent who plays the minimax strategy because X must also learn to take advantage of mistakes (non-minimax play by the opponent).

When two minimax strategies are played against each other in this particular game, the payoff is 12, which is the value of this game. A minimax strategy typically takes advantage of non-minimax play by the other player.

A hit for this problem (which we also sometimes call a minimax hit in problems involving games) is the number of fitness cases (out of 4 for player X or 8 for player O) where the strategy being tested achieves a payoff at least as good as the minimax strategy.

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

In one run, the best single individual game-playing strategy for player X in generation 6 was

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

Note that this strategy for player X is a composition of the four functions (`CXM1`, `COM1`, `CXM2`, `COM2`) and two terminals (`L` and `R`) and that it returns a value of either `L` or `R`.

This strategy for player X simplifies to

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

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

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

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

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

Note that we needed the minimax strategy for player O to serve as the environment for evolving this game-playing strategy for player X.

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

In one run of the genetic programming paradigm, the best single individual strategy for player O in generation 9 had a fitness of 52 and scored 8 hits. This minimax O strategy was

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

This strategy for player O simplifies to

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

Note that, in simplifying this strategy, we inserted the filler symbol # to indicate a situation that can never arise.

Note that we needed the minimax strategy for player X to serve as the environment for evolving this game-playing strategy for player O.

## 19. EMERGENT BEHAVIOR FOR AN ANT COLONY

The repetitive application of seemingly simple rules can lead to complex overall “emergent behavior” (Forrest[37]). Examples of complex overall behavior that emerges from relatively simple rules occur in the study of cellular automata, Lindenmayer systems, fractals, and chaos as well as in nature.

Emergent functionality, according to Steels[38], means that an overall function is not achieved in the familiar way by a hierarchical system of components, but, instead, indirectly by the interaction of more primitive components with the world and among themselves.

Emergent functionality is one of the main themes of research in artificial life (Langton et. al.[39], Farmer et. al.[40]).

One avenue of work in emergent behavior involves researchers writing sets of rules that produce the desired complex overall behavior. In this section, we use the genetic programming paradigm to evolve the sets of rules.

In this section, we genetically breed a computer program controlling the behavior of an individual ant which, when simultaneously executed in parallel by all the ants in an ant colony, causes the emergence of interesting higher level collective behavior for the colony as a whole.

In particular, the goal is to genetically evolve a common computer program governing the behavior of the individual ants such that the collective behavior of the ants consists of efficient transportation of food to the colony. In nature, when an ant discovers food, it deposits a trail of chemicals (called pheromones) as it returns to the nest with the food. The pheromonal cloud (which dissipates over time) aids other ants in efficiently locating and exploiting the food source (Travers and Resnick[41], Resnick[42]). An equivalent problem involves robots on the moon bringing rock samples back to the space ship (Steels[38]).

In this problem, 144 pellets of food are piled eight deep in two 3-by-3 piles located in a 32-by-32 toroidal area. There are 20 ants. The state of each ant consists of its position and the direction it is facing (out of eight possible directions). Each ant initially starts at the nest and faces in a random direction. Each ant in the colony is governed by a common computer program associated with the colony.

The following nine operations are available in this problem:

- MOVE-RANDOM randomly changes the direction in which an ant is facing and then moves the ant two steps in the new direction.
- MOVE-TO-NEST moves the ant one step in the direction of the nest. This implements the gyroscopic ability of ants to navigate back to their nest.
- PICK-UP picks up food (if any) at the current position of the ant if the ant is not already carrying food.
- DROP-PHEROMONE drops pheromones at the current position of the ant (if the ant is carrying food). The pheromones

immediately forms a 3-by-3 cloud around the drop point. The cloud decays over a period of time.

- IF-FOOD-HERE is a two-argument function that executes its first argument if there is food at the ant's current position and, otherwise, executes the second (else) argument.
- IF-CARRYING-FOOD is a similar two-argument function that tests whether the ant is currently carrying food.
- MOVE-TO-ADJACENT-FOOD-ELSE is a one-argument function that allows the ant to test for immediately adjacent food and then move one step towards it. If food is present in more than one adjacent position, the ant moves to the position requiring the least change of direction. If no food is adjacent, the "else" clause of this function is executed.
- MOVE-TO-ADJACENT-PHEROMONE-ELSE is a function similar to the above based on adjacency of pheromones.
- PROGN is the LISP connective function that executes its arguments in sequence

Each of the 20 ants in a given colony executes the colony's common computer program at each time step. The execution of the common program is done serially for each ant for a given time step. Thus, the action of one ant can alter the state of the system for other ants (e.g. by picking up food or dropping pheromones). Since the ants initially face in random directions, make random moves, and encounter a changing pattern of food and pheromones created by the activities of other ants, the 20 ants almost always have different states and pursue different trajectories.

The first major step in using the genetic programming paradigm is to identify the set of terminals. The terminal set for this problem is

```
T = {MOVE-RANDOM, MOVE-TO-NEST, PICK-UP, DROP-
     PHEROMONE}
```

The second major step in using the genetic programming paradigm is to identify the set of functions. The function set for this problem is

```
F = {IF-FOOD-HERE, IF-CARRYING-FOOD, MOVE-TO-
     ADJACENT-FOOD-ELSE, MOVE-TO-ADJACENT-
     PHEROMONE-ELSE, PROGN}
```

The third major step in using the genetic programming paradigm is to identify the fitness function. The raw fitness of a computer program is measured by how many of the 144 food pellets are transported to the nest within the allotted time (which limits both the total number of time steps and the total number of operations which any one ant can execute) when all the ants execute the given program.

Note that there are no explicit fitness cases in this problem. There are sufficient ants (each with their own initial facing direction) so that the fitness cases can be implicit.

Mere random motion by the 20 ants in a colony will, on average, only bring the ants into contact with about 56 of the 144 food pellets within the allotted time. Of course, the task is substantially more complicated than merely coming in contact with food at random. After coming into contact with food, the ants must pick up the food and then move towards the colony. Moreover, even this sequence of behavior is not sufficient to efficiently solve the problem in any reasonable amount of time. When ants come in contact with food, they must do something which makes the task easier than mere random search thereafter. In particular, ants which come in contact with food must also establish a pheromonal trail as they carry the food back to the colony. This allows other ants to use the existence of the pheromonal trail as a guide to the location of the food. Of course, in addition, all ants must always be on the lookout for such pheromonal trails and must, if they are not already carrying food, follow such trails to the food when they encounter such trails.

In a typical run, 93% of the random computer programs in the initial random generation did not transport even one of the 144 food pellets to the nest within the allotted time. About 3% of these initial random programs transported only one of the 144 pellets. Even the best single computer program of the random computer programs created in the initial generation successfully transported only 41 pellets (i.e. only about 2 per ant).

As the genetic programming paradigm is run, the population as a whole and its best single individual both generally improve from generation to generation.

In one run, the best single individual in the population on generation 10 scored 54; the best-of-generation individual on generation 20 scored 72; the best-of-generation individual on generation 30 scored 110; the best-of-generation individual on generation 35 scored 129; the best-of-generation individual on generation 37 scored 142; and, the best-of-generation individual scored 144 (i.e. 100% ) on generation 38.

Figure 1.57 shows the progressive improvement during that run of the worst single individual in the population, the average standardized fitness of the population, and the best-of-generation individual.

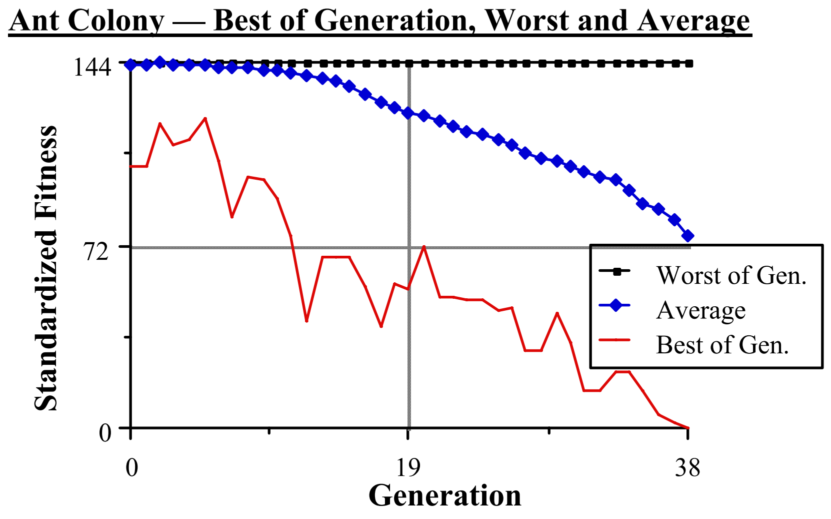


*Figure 1.57 Standardized fitness for worst-of-generation individual, average standardized fitness for population, and standardized fitness for best-of-generation individual for emergent behavior in the ant colony*

On generation 38, a program emerged which allows the 20 ants to successfully transport all 144 food pellets to the nest within the allotted time. This 100% fit program is shown below:

```
(PROGN (PICK-UP) (IF-CARRYING-FOOD (PROGN (MOVE-TO-
ADJACENT-PHEROMONE-ELSE (MOVE-TO-ADJACENT-FOOD-ELSE
(MOVE-TO-ADJACENT-FOOD-ELSE (MOVE-TO-ADJACENT-FOOD-
ELSE (PICK-UP))))) (PROGN (PROGN (PROGN (PROGN (MOVE-
TO-ADJACENT-FOOD-ELSE (PICK-UP)) (PICK-UP)) (PROGN
(MOVE-TO-NEST) (DROP-PHEROMONE))) (PICK-UP)) (PROGN
(MOVE-TO-NEST) (DROP-PHEROMONE)))) (MOVE-TO-ADJACENT-
FOOD-ELSE (IF-CARRYING-FOOD (PROGN (PROGN (DROP-
PHEROMONE) (MOVE-TO-ADJACENT-PHEROMONE-ELSE (IF-
CARRYING-FOOD (MOVE-TO-ADJACENT-FOOD-ELSE (PICK-UP))
(MOVE-TO-ADJACENT-FOOD-ELSE (PICK-UP))))) (MOVE-TO-
NEST)) (IF-FOOD-HERE (PICK-UP) (IF-CARRYING-FOOD
(PROGN (IF-FOOD-HERE (MOVE-RANDOM) (IF-CARRYING-FOOD
(MOVE-RANDOM) (PICK-UP))) (DROP-PHEROMONE)) (MOVE-TO-
ADJACENT-PHEROMONE-ELSE (MOVE-RANDOM)))))))
```

The 100% fit program above is essentially equivalent to the simplified program below:

```
1 (PROGN (PICK-UP)
2     (IF-CARRYING-FOOD
3       (PROGN (MOVE-TO-ADJACENT-PHEROMONE-ELSE
4              (MOVE-TO-ADJACENT-FOOD-ELSE (PICK-UP)))
5               (MOVE-TO-ADJACENT-FOOD-ELSE (PICK-UP))
6                (MOVE-TO-NEST)
7                (DROP-PHEROMONE)
8                (MOVE-TO-NEST)
9                (DROP-PHEROMONE))
10       (MOVE-TO-ADJACENT-FOOD-ELSE
11           (IF-FOOD-HERE
12             (PICK-UP)
13             (MOVE-TO-ADJACENT-PHEROMONE-ELSE
14               (MOVE-RANDOM))))))
```

This computer program is a prioritized sequence of conditional behaviors that work together to solve the problem. First, the computer program causes the ant to pick up any food it may encounter. Failing that, the second priority established by this conditional sequence causes the ant to follow a previously established pheromonal trail. And, failing that, the third priority of this conditional sequence causes the ant to move at random.

This simplified program prioritizes the activities of the ant and can be interpreted as follows: The ant begins (line 1) by picking-up the food, if any, located at the ant's current position. If the test on line 2 determines that the ant is now carrying food, then lines 3 through 9 are executed. Otherwise, lines 10 through 14 are executed.

Line 3 moves the ant to the adjacent pheromones (if any). If there is no adjacent pheromone, line 4 moves the ant to the adjacent food (if any). In view of the fact that the ant is already carrying food, these two potential moves on lines 3 and 4 generally distract the ant from the most direct return to the nest and therefore merely reduce efficiency. Line 5 is a similar distraction. Note that the PICK-UP operations on lines 4 and 5 are redundant since the ant is already carrying a food pellet. Given that the ant is already carrying food, the sequence of MOVE-TO-NEST on line 6 and DROP-PHEROMONE on line 7 is the winning combination that establishes the pheromone trail as the ant moves towards the nest with the food. This move sequence is repeated redundantly in lines 8 and 9. The establishment of the pheromone trail between the pile of food and the nest is an essential part of efficient collective behavior for exploiting the food source.

Lines 10 through 13 apply when line 2 determines that the ant is not carrying food. Line 10 moves the ant to adjacent food (if any). If there is no adjacent food but there is food at the ant's current position (line 11), the ant picks up the food (line 12). On the other hand, if there is no food at the ant's current position (line 13), the ant moves towards any adjacent pheromones (if any). If there are no adjacent pheromones, the ant moves randomly (line 14).

The collective behavior of the ant colony governed by the above 100% fit program above can be visualized as a series of major phases. The first phase occurs when the ants have just emerged from the nest and are randomly searching for food.

Figure 1.58 represents time step 3 of the execution of the 100% fit program above. The two 3-by-3 piles of food are shown in black in the western and northern parts of the grid. The nest is indicated by nine + signs slightly southeast of the center of the grid. The ants are shown in gray with their facing direction indicated.

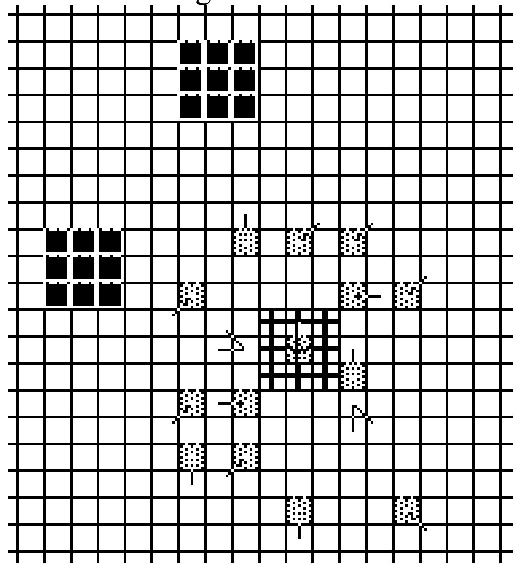


*Figure 1.58 The First phase of the emergent behavior in an ant colony*

The second phase occurs when some ants have discovered some food, have picked up the food, and have started back towards the nest dropping pheromones as they go. The beginnings of the pheromonal clouds around both the western and northern piles of food are shown in Figure 1.59 (representing time step 12).

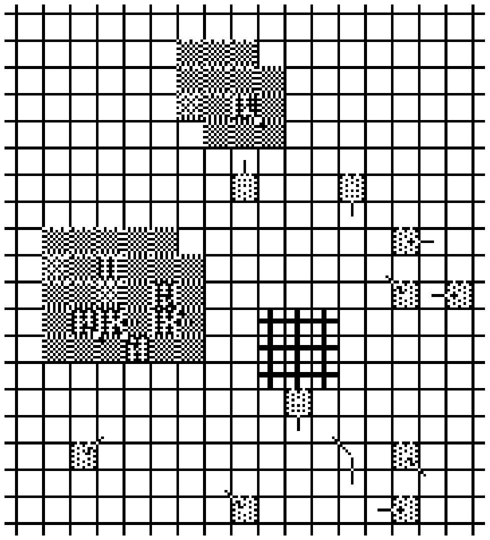


*Figure 1.59  The Second phase of the emergent behavior in an ant colony*

The third phase occurs when pheromonal trails have been established linking both piles of food with the colony. The first two (of the 144) food pellets have just reached the nest in Figure 1.60 (representing time step 15).

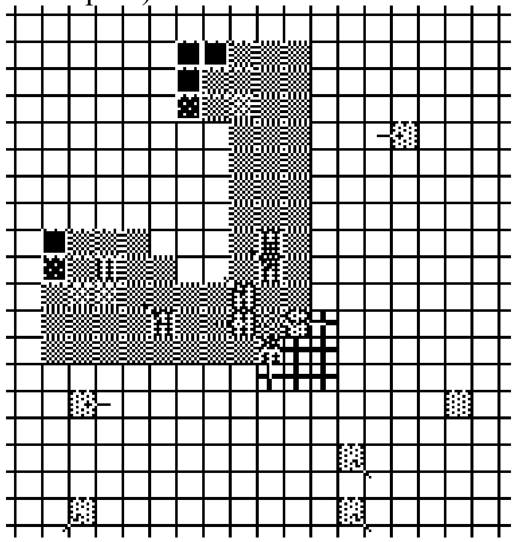


*Figure 1.60  The Third phase of the emergent behavior in an ant colony*

This third phase persists for some time as the ants transport the bulk of the food from the piles to the colony.

Figure 1.61 shows the premature (and temporary) disintegration of the pheromonal trail connecting the northern pile of food with the nest while food still remains in the northern pile. The pheromonal trail connecting the western pile of food with the nest is still intact. 118 of the 144 food pellets have been transported to the nest at this point (at time step 129).

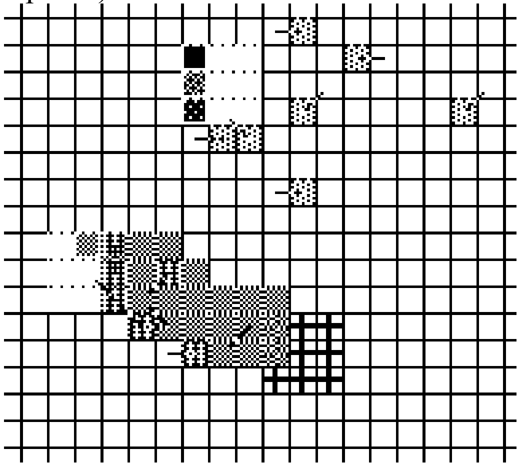


*Figure 1.61 Premature disintegration of the pheromonal trail in the problem of emergent behavior in an ant colony*

Figure 1.62 (representing time step 152) shows the western pile has been entirely cleared by the ants and the pheromonal trail connecting it to the nest has already dissipated. The former location of the western pile is shown as a blank white area. 136 of the 144 food pellets have been transported to the nest at this point. The pheromonal trail connecting the nest to the northern pile (with 8 remaining food pellets) has been reestablished.

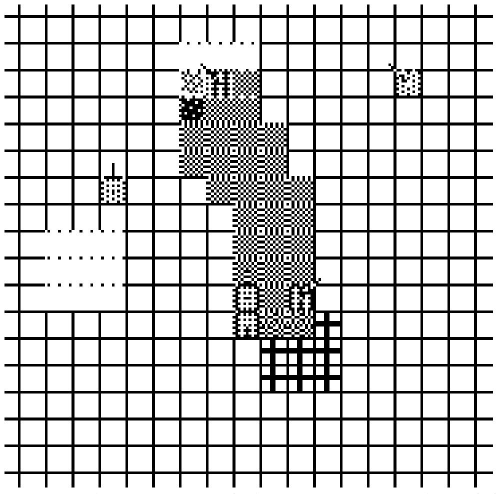


*Figure 1.62 Exhaustion of the western pile of food in the problem of emergent behavior in an ant colony*

Shortly thereafter, the run ends with all 144 food pellets in the nest.

A visualization of the application of the genetic programming paradigm to this problem (as well as to planning, empirical discovery, inverse kinematics, game playing, and the Boolean 11-multiplexer problems) can be viewed in the *Artificial Life II Video Proceedings* videotape [Koza and Rice[43]].

## 20. ADDITIONAL EXAMPLES

The genetic programming paradigm can be applied in many additional problem domains.

For example, induction of decision trees and concept formation can be done using the genetic programming paradigm (Koza[44]).

In addition, when the LISP S-expressions return more than one value, even more complex structures can be evolved using the genetic programming paradigm. One example is the simultaneous discovery of both the architecture and weights for a neural network (Koza and Rice[45]).

Also, more than one population can be evolved at the same time to solve problems such as simultaneously evolving minimax strategies for both players in a game (Koza[46, 47]).

Additional information and examples can be found in Koza[28, 48].

## 21. CONCLUSIONS

We have shown that many seemingly different problems in machine learning and artificial intelligence can be viewed as requiring the discovery of a computer program that produces some desired output for particular inputs. We have also shown that the recently developed genetic programming paradigm described herein provides a way to search for a highly fit individual computer program.

## 22. ACKNOWLEDGEMENTS

James Rice of the Knowledge Systems Laboratory created all of the graphics herein, aided in programming many of the problems presented herein, and provided numerous useful comments on this chapter.

Lawrence Davis, John Holland, Martin Keane, John Perry, Rick Riolo, Tom Westerdale, Stewart Wilson provided numerous useful comments and discussions on various drafts of the material in this chapter.

## 23. REFERENCES

1. Holland, John H. *Adaptation in Natural and Artificial Systems.* Ann Arbor, MI: University of Michigan Press 1975.
2. Goldberg, David E. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley l989.
3. Davis, Lawrence (editor) *Genetic Algorithms and Simulated Annealing*. London: Pittman l987.
4. Davis, Lawrence. *Handbook of Genetic Algorithms*. New York: Van Nostrand Reinhold.1991.
5. Schaffer, J. D. (editor). *Proceedings of the Third International Conference on Genetic Algorithms.* San Mateo, CA: Morgan Kaufmann Publishers Inc. 1989.
6. Rawlins, Gregory (editor). *Proceedings of Workshop on the Foundations of Genetic Algorithms and Classifier Systems. Bloomington, Indiana. July 15-18, 1990.* San Mateo, CA: Morgan Kaufmann 1991.
7. Belew, Rik and Booker, Lashon (editors) *Proceedings of the Fourth International Conference on Genetic Algorithms*. San Mateo, Ca: Morgan Kaufmann 1991.

8. Davis, Lawrence and Steenstrup, M. Genetic algorithms and simulated annealing: An overview. In Davis, Lawrence (editor) *Genetic Algorithms and Simulated Annealing* London: Pittman l987.
9. De Jong, Kenneth A. Learning with genetic algorithms: an overview. *Machine Learning*, 3(2), 121-138, 1988.
10. Smith, Steven F. *A Learning System Based on Genetic Adaptive Algorithms*. PhD dissertation. Pittsburgh, PA University of Pittsburgh 1980.
11. 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.
12. Holland, John H, Holyoak, K.J., Nisbett, R.E., and Thagard, P.A. *Induction: Processes of Inference, Learning, and Discovery*. Cambridge, MA: MIT Press l986.
13. Wilson, Stewart. W. Classifier Systems and the animat problem. *Machine Learning*, 3(2), 199-228, 1987.
14. Wilson, Stewart W. Bid competition and specificity reconsidered. *Journal of Complex Systems.* 2(6), 705-723, 1988.
15. Forrest, Stephanie. *Parallelism and Programming in Classifier Systems*. London: Pittman 1991.
16. Wilson, Stewart. W. Hierarchical credit allocation in a classifier system. *Proceedings of the Tenth International Joint Conference on Artificial Intelligence.* San Mateo, CA: Morgan Kaufmann, 217-220, 1987.
17. Goldberg, David E., Korb, Bradley, and Deb, Kalyanmoy. Messy genetic algorithms:Motivation, Analysis, and First Results. *Complex Systems*. 3(5) October 1989. Pages 493-530.
18. Cramer, Nichael Lynn. A representation for the adaptive generation of simple sequential programs. In Grefenstette, John J.(editor). *Proceedings of an International Conference on Genetic Algorithms and Their Applications*. Hillsdale, NJ: Lawrence Erlbaum Associates l985.
19. 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.

20. 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.
21. 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.
22. Lenat, Douglas B. AM: *An Artificial Intelligence Approach to Discovery in Mathematics as Heuristic Search.* PhD Dissertation. Computer Science Department. Stanford University. 1976.
23. 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.
24. Lenat, Douglas B. and Brown, John Seely. Why AM and EURISKO appear to work. *Artificial Intelligence*. 23 (1984). 269-294.
25. Jefferson, David et. al. The Genesys System: Evolution as a theme in artificial life. In Farmer, Doyne, Langton, Christopher, Rasmussen, S., and Taylor, C. (editors) *Artificial Life II, Santa Fe Institute Studies in the Sciences of Complexity*. Volume XI. Redwood City, CA: Addison-Wesley. 1991.
26. Anderson, Stuart L. Random number generators on vector supercomputers and other advanced architectures. *SIAM Review*. 32(2). Pages 221-251. June 1990.
27. Knuth, Donald E. *The Art of Computer Programming*. Volume 2. Reading, MA: Addison-Wesley 1981.
28. Koza, John R. *Genetic Programming*. Cambridge, MA: MIT Press 1991. (Forthcoming).
29. Citibank, N. A. *CITIBASE: Citibank Economic Database (Machine Readable Magnetic Data File), 1946-Present.* New York: Citibank N.A. 1989.
30. Hallman, Jeffrey J., Porter, Richard D., Small, David H. *M2 per Unit of Potential GNP as an Anchor for the Price Level.* Washington,DC: Board of Governors of the Federal Reserve System. Staff Study 157, April 1989.

31. 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.
32. Genesereth, Michael R. and Nilsson, Nils J. *Logical Foundations of Artificial Intelligence*. Los Altos, CA: Morgan Kaufmann 1987.
33. Widrow, Bernard. The original adaptive neural net broom balancer. *IEEE International Symposium on Circuits and Systems*. Vol. 2. 1987.
34. Anderson, Charles W. Learning to control and inverted pendulum using neural networks. *IEEE Control Systems Magazine*. 9(3). Pages 3l-37. April l989.
35. 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 15-19, 1990*. Volume I. Hillsdale, NJ: Lawrence Erlbaum 1990.
36. 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*. *Antibes, June, 1990.* Pages 47-56. Berlin: Springer-Verlag, 1990.
37. Forrest, Stephanie (editor). *Emergent Computation: Self-organizing, Collective, and Cooperative Computing Networks*. Cambridge, MA: MIT Press 1990. Also in Physica D 1990.
38. Steels, Luc. Towards a theory of emergent functionality. 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. Cambridge, MA: MIT Press 1991.
39. Langton, Christopher G. *Artificial Life II, Santa Fe Institute Studies in the Sciences of Complexity*. Volume VI. Redwood City, CA: Addison-Wesley. 1989.
40. Farmer, Doyne, Langton, Christopher, Rasmussen, S., and Taylor, C. (editors) *Artificial Life II, SFI Studies in the Sciences of Complexity*. Volume XI. Redwood City, CA: Addison-Wesley 1991.

41. Travers, Michael and Resnick, Mitchel. Behavioral Dynamics of an Ant Colony: Views from Three Levels. Videotape. Cambridge, MA: MIT Media Laboratory 1990.
42. Resnick, Mitchel. Animal simulations with *Logo: massive parallelism for the masses. 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. Cambridge, MA: MIT Press 1991.
43. 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.
44. 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.* Berlin: Springer-Verlag. 1991.
45. 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.
46. 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. Cambridge, MA: MIT Press 1991.
47. 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. Redwood City, CA: Addison-Wesley 1991.
48. 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: Morgan Kaufman 1989.