Statistics and Computing (1994) **4**, 87–112

Genetic programming as a means for programming computers by natural selection

JOHN R. KOZA

Computer Science Department, Stanford University, Stanford, CA 94305, USA

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 genetic programming, 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. Genetic programming is illustrated via an example of machine learning of the Boolean 11-multiplexer function and symbolic regression of the econometric exchange equation from noisy empirical data.

Hierarchical automatic function definition enables genetic programming to define potentially useful functions automatically and dynamically during a run, much as a human programmer writing a complex computer program creates subroutines (procedures, functions) to perform groups of steps which must be performed with different instantiations of the dummy variables (formal parameters) in more than one place in the main program. Hierarchical automatic function definition is illustrated via the machine learning of the Boolean 11-parity function.

Keywords: Genetic programming, genetic algorithm, crossover, hierarchical automatic function definition, symbolic regression, Boolean 11-multiplexer, econometric exchange equation, Boolean 11-parity

1. Introduction and overview

Computer programs are among the most complex and intricate structures created by human beings. They are usually written line by line by applying human knowledge and intelligence to the problem at hand. Writing a computer program is usually difficult. Indeed, one of the central questions in computer science (attributed to Arthur Samuel in the 1950s) is

> How can computers learn to solve problems without being explicitly programmed? In other words, how can computers be made to do what is needed to be done, without being told exactly how to do it?

In the natural world, complex and intricate structures do not arise via explicit design and programming or from the application of human intelligence. Instead, complex and successful organic structures evolve over a period of time as the consequence of Darwinian natural selection and the creative effects of sexual recombination (genetic crossover) and mutation. Complex structures evolve in nature as a consequence of a fitness metric applied by the problem environment because structures that are more fit in grappling with their environment survive and reproduce at a higher rate.

The question arises as to whether an analogue of natural selection and genetics can be applied to the problem of creating a program that enables a computer to solve a problem. That is, can complex computer programs be created, not via human intelligence, but by applying a fitness measure appropriate to the problem environment?

Such a process of genetical breeding of computer programs might start with a primordial ooze consisting of a population of hundreds or thousands of randomly created computer programs of various randomly determined sizes and shapes. In such a process, each program in the population would be observed as it tries to grapple with its environment—that is, to solve the problem at hand. A value would then be assigned to each program, reflecting how fit it is in solving the problem at hand. We might then allow a program in the population to survive to a later generation of the process with a probability proportionate to its observed fitness. Additionally, we might also select pairs of programs from the population with a probability proportionate to their observed fitness and create new offspring by recombining subprograms from them at random. We would apply the above steps to the population of programs over a number of generations.

Anyone who has ever written and debugged computer programs and has experienced their brittle, highly non-linear, and perversely unforgiving nature will probably be understandably sceptical about the proposition that the biologically motivated process sketched above could possibly produce a useful computer program. However, in this article we present a number of examples from various fields supporting the surprising and counter-intuitive notion that computers can indeed be programmed by means of natural selection. We will show, via examples, that the recently developed genetic programming paradigm provides a way to search the space of all possible programs to find a function which solves, or approximately solves, a problem.

2. Background on genetic algorithms and genetic programming

John Holland's pioneering book *Adaptation in Natural and Artificial Systems* described how the evolutionary process in nature can be applied to artificial systems using the genetic algorithm operating on fixed-length character strings (Holland, 1975). Holland demonstrated that a population of fixed-length character strings (each representing a proposed solution to a problem) can be genetically bred using the Darwinian operation of fitness proportionate reproduction and the genetic operation of recombination. The recombination operation combines parts of two chromosome-like fixed-length character strings, each selected on the basis of their fitness, to produce new offspring strings. Holland established, among other things, that the genetic algorithm is a near optimal approach to adaptation in that it maximizes expected overall average payoff when the adaptive process is viewed as a multi-armed slot machine problem requiring an optimal allocation of future trials given currently available information. The genetic algorithm has proven successful at searching non-linear multidimensional spaces in order to solve, or approximately solve, a wide variety of problems (Goldberg, 1989; Davis, 1987; 1991; Davidor, 1991; Michalewicz, 1992). Recent conference proceedings provide an overview of current work in the field (Schaffer, 1989; Forrest, 1990; Belew and Booker, 1991; Rawlins, 1991; Mayer and Wilson, 1991; Schwefel and Maenner, 1991; Langton *et al.*, 1992; Whitley, 1992).

Representation is a key issue in genetic algorithm work because genetic algorithms directly manipulate a coded chromosomal representation of the problem. The representation scheme can therefore severely limit the window by which the system observes its world. On the other hand, 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. The need for increasing the complexity of the structures undergoing adaptation using the genetic algorithm has been reflected by considerable work over the years in that direction (Smith, 1980; Cramer, 1985; Holland, 1986; Holland *et al.*, 1986; Wilson, 1987*a*,*b*; Fujiki and Dickinson, 1987; Goldberg *et al.*, 1989).

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

Genetic programming provides a way to find a computer program of unspecified size and shape to solve, or approximately solve, a problem. The book *Genetic Programming: On the Programming of Computers by Means of Natural Selection* (Koza, 1992*a*) describes genetic programming in detail. A videotape visualization of applications of genetic programming can be found in the *Genetic Programming: The Movie* (Koza and Rice, 1992*a*). See also Koza (1992*b*).

3. Overview of genetic programming

Genetic programming continues the trend of dealing with the problem of representation in genetic algorithms by increasing the complexity of the structures undergoing adaptation. In particular, the individuals in the population in genetic programming are hierarchical compositions of primitive functions and terminals appropriate to the particular problem domain. The set of primitive functions used typically includes arithmetic operations, mathematical functions, conditional logical operations, and domain-specific functions. The set of terminals used typically includes inputs appropriate to the problem domain and various numeric constants.

The compositions of primitive functions and terminals described above correspond directly to the computer programs found in programming languages such as LISP (where they are called symbolic expressions or S-expressions). An S-expression can be represented as a rooted, point-labelled tree with ordered branches in which the root and other internal points of the tree are labelled with functions and in which the external points of the tree are labelled with terminals. In fact, these compositions correspond directly to the parse tree that is internally created by the compilers of most programming languages. Thus, genetic programming views the search for a solution to a problem as a search in the space of all possible compositions of functions that can be recursively composed of the available primitive functions and terminals.

Of course, virtually any problem in artificial intelligence, symbolic processing, and machine learning can be viewed as requiring discovery of a computer program that produces some desired output for particular inputs. The process of solving these problems can be reformulated as a search for a highly 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 the fittest individual computer program. In particular, the search space is the space of all possible computer programs composed of functions and terminals appropriate to the problem domain. Genetic programming provides a way to search for this fittest individual computer program.

In genetic programming, populations of hundreds or thousands of computer programs are genetically bred. This breeding is done using the Darwinian principle of survival and reproduction of the fittest along with a genetic recombination (crossover) operation appropriate for mating computer programs. As will be seen, a computer program that solves (or approximately solves) a given problem may emerge from this combination of Darwinian natural selection and genetic operations.

Genetic programming 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, or 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. The creation of this initial random population is, in effect, a blind random search of the search space of the problem.

Each individual computer program in the population is measured in terms of how well it performs in the particular problem environment. This measure is called the *fitness measure*.

The nature of the fitness measure varies with the problem. For many problems, fitness is naturally measured by the error produced by the computer program. The closer this error is to zero, the better the computer program. If one is trying to find a good randomizer, the fitness of a given computer program might be measured via entropy. The higher the entropy, the better the randomizer. If one is trying to recognize patterns or classify examples, the fitness of a particular program might be the number of examples (instances) it handles correctly. The more examples correctly handled, the better. In a problem of optimal control, the fitness of a computer program may be the amount of time or fuel or money required to bring the system to a desired target state. The smaller the amount of time or fuel or money, the better. For some problems, fitness may consist of a combination of factors such as correctness, parsimony, or efficiency.

Typically, each computer program in the population is run over a number of different *fitness cases* so that its fitness is measured as a sum or an average over a variety of representative different situations. These fitness cases sometimes represent a sampling of different values of an independent variable or a sampling of different initial conditions of a system. For example, the fitness of an individual computer program in the population may be measured in terms of the sum of the squares of the differences between the output produced by the program and the correct answer to the problem. This sum may be taken over a sampling of different inputs to the program. The fitness cases may be chosen at random or may be structured in some way.

The computer programs in generation 0 will have exceedingly poor fitness. None the less, some individuals in the population will turn out to be somewhat fitter than others. These differences in performance are then exploited. The Darwinian principle of reproduction and survival of the fittest and the genetic operation of sexual recombination (crossover) are used to create a new offspring population of individual computer programs from the current population of programs. The reproduction operation involves selecting on the basis of fitness (i.e. the fitter the program, the more likely it is to be selected), a computer program from the current population of programs, and allowing it to survive by copying it into the new population.

The genetic process of sexual reproduction between two parental computer programs is used to create new offspring computer programs from two parental programs selected on the basis of fitness. The parental programs are typically of different sizes and shapes. The offspring programs are composed of subexpressions (subtrees, subprograms, subroutines, building blocks) from their parents. These offspring programs are typically of different sizes and shapes than their parents. Intuitively, if two computer programs are somewhat effective in solving a problem, then some of their parts probably have some merit. By recombining randomly chosen parts of somewhat effective programs, we may produce new computer programs that are even fitter in solving the problem. For example, consider the following computer program (LISP symbolic expression):

$$(+\underline{(*0.234Z)}(-X0.789)),$$

which we would ordinarily write as

$$0.234Z + X - 0.789.$$

This program takes two inputs (X and Z) and produces a floating-point output. In the prefix notation used, the multiplication function * is first applied to the terminals 0.234 and Z to produce an intermediate result. Then, the subtraction function – is applied to the terminals X and 0.789 to produce a second intermediate result. Finally, the addition function + is applied to the two intermediate results to produce the overall result.

Also, consider a second program:

$$(*(*ZY)\underline{(+Y(*0.314Z))}),$$

which is equivalent to

$$ZY(Y + 0.314Z).$$

In Fig. 1, these two programs are depicted as rooted, point-labelled trees with ordered branches. Internal points (i.e. nodes) of the tree correspond to functions (i.e. operations) and external points (i.e. leaves, endpoints) correspond to terminals (i.e. input data). The numbers beside the function and terminal points of the tree appear for reference only. The crossover operation creates new offspring by exchanging subtrees (i.e. sublists, subroutines, subprocedures) between the two parents.

Assume that the points of both trees are numbered in a depth-first way starting at the left. Suppose that the point number 2 (out of 7 points of the first parent) is randomly selected as the crossover point for the first parent and that the point number 5 (out of 9 points of the second parent) is randomly selected as the crossover point of the second parent. The crossover points in the trees above are therefore the * in the first parent and + in the second parent. The two crossover fragments are the two sub-trees shown in Fig. 2. These two crossover fragments correspond to the underlined subprograms (sublists) in the two parental computer programs. The two offspring resulting from crossover are

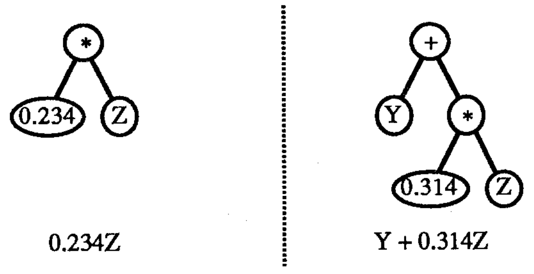

Fig. 2. *Two crossover fragments*

$$(+\underline{(+Y(*0.314Z))}(-X0.789))$$

and

$$(*(*ZY)\underline{(*0.234Z)}).$$

The two offspring are shown in Fig. 3.

Thus, crossover creates new computer programs using parts of existing parental programs. Because entire subtrees are swapped, this crossover operation always produces syntactically and semantically valid programs as offspring regardless of the choice of the two crossover points. Because programs are selected to participate in the crossover operation with a probability proportional to fitness, crossover allocates future trials to areas of the search space represented by programs containing parts from promising programs.

After the operations of reproduction and crossover are performed on the current population, the population of offspring (i.e. the new generation) replaces the old population (i.e. the old generation). Each individual in the new population of computer programs is then measured for fitness, and the process is repeated over many generations.

At each stage of this highly parallel, locally controlled,

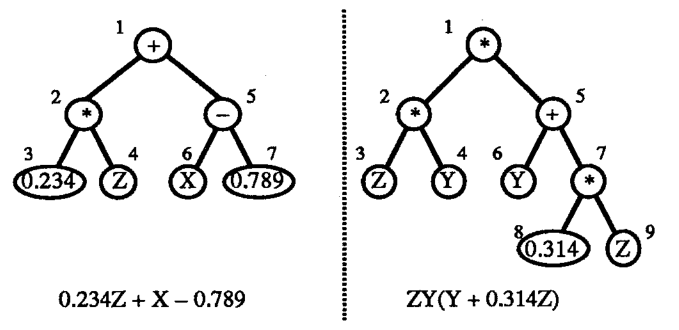

Fig. 1. *Two parental computer programs*

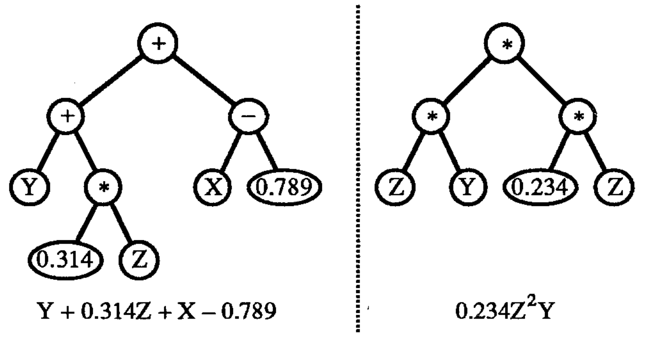

Fig. 3. *Two offspring*

decentralized process, the state of the process will consist only of the current population of individuals. The force driving this process consists only of the observed fitness of the individuals in the current population in grappling with the problem environment.

As will be seen, this algorithm will produce populations of computer programs which, over many generations, tend to exhibit increasing average fitness in dealing with their environment. In addition, these populations of computer programs can rapidly and effectively adapt to changes in the environment. Typically, the best individual that appeared in any generation of a run (i.e. the best-so-far individual) is designated as the result produced by genetic programming.

The hierarchical character of the computer programs that are produced is an important feature of genetic programming. The results of genetic programming are inherently hierarchical. In many cases the results produced by genetic programming are default hierarchies, prioritized hierarchies of tasks, or hierarchies in which one behaviour subsumes or suppresses another.

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

Another important feature of genetic programming is the absence or relatively minor role of preprocessing of inputs and postprocessing of outputs. The inputs, intermediate results, and outputs are typically expressed directly in terms of the natural terminology of the problem domain. The computer programs produced by genetic programming consist of functions that are natural for the problem domain.

Finally, the structures undergoing adaptation in genetic programming are active. They are not passive chromosomal encodings of the solution to the problem. Instead, given a computer on which to run, the structures in genetic programming are active program structures that are capable of being executed in their current form.

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

1. Generate an initial population of random computer programs composed of the primitive functions and terminals of the problem.
2. Iteratively perform the following substeps until the termination criterion for the run has been satisfied:
   (a) Execute each program in the population so that a fitness measure indicating how well the program solves the problem can be computed for the program.
   (b) Create a new population of programs by selecting program(s) in the population with a probability based on fitness (i.e. the fitter the program, the more likely it is to be selected) and then applying the following primary operations:
      (i) *Reproduction*: Copy an existing program to the new population.
      (ii) *Crossover*: Create two new offspring programs for the new population by genetically recombining randomly chosen parts of two existing programs.
3. The single best computer program in the population produced during the run is designated as the result of the run of genetic programming. This result may be a solution (or approximate solution) to the problem.

Figure 4 is a flowchart for genetic programming. The index i refers to an individual in the population of size M. The variable GEN is the number of the current generation. The box labelled 'evaluate fitness of each individual in the population' typically consumes the vast majority of computer resources.

In the remainder of this article, we illustrate genetic programming with several examples chosen to illustrate various different categories of problems, namely:

- symbolic regression of a Boolean-valued function;
- symbolic regression of noisy numeric-valued empirical data;
- a multidimensional control problem;
- a classification problem;
- a robotics problem;
- a problem employing hierarchical automatic function definition.

4. Symbolic regression–11-multiplexer

The problem of symbolic function identification (symbolic regression) requires developing a composition of terminals and functions that can return the correct value of the function after seeing a finite sampling of combinations of the independent variable associated with the correct value of the dependent variable. The problem of machine learning of a Boolean function is a special case of symbolic regression in which the independent variables are Boolean-valued, the functions being composed are Boolean functions, and the dependent variable is Boolean-valued.

The problem of learning the Boolean 11-multiplexer function will serve to show the interplay in genetic programming of

- the genetic variation inevitably created in the initial random generation;

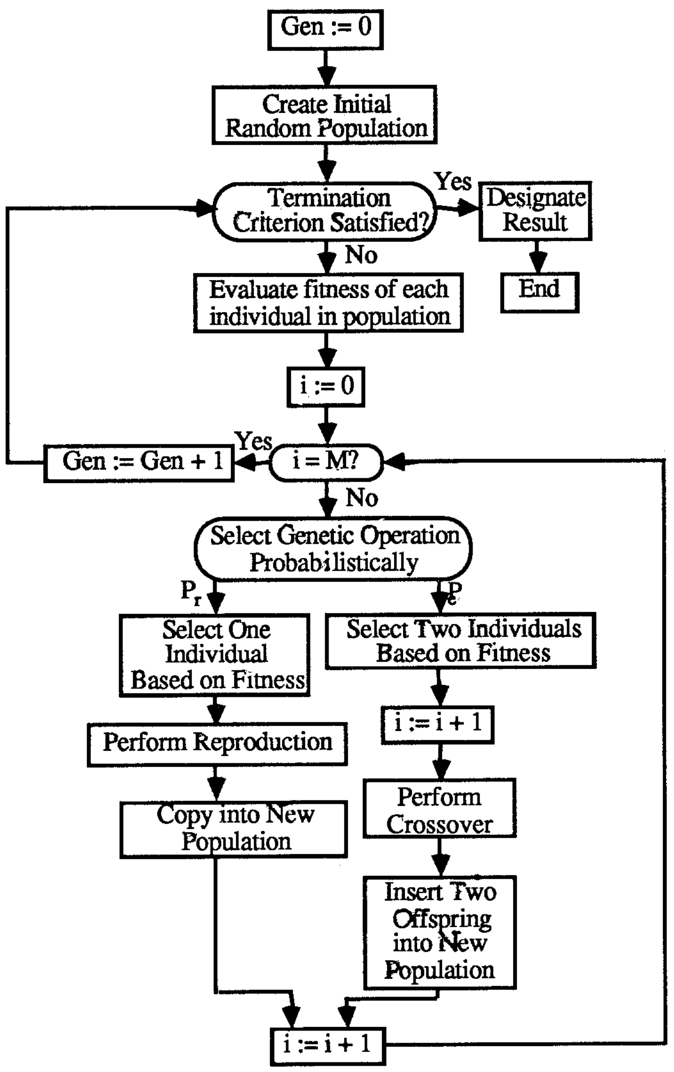
Gen := 0
Create Initial Random Population
Termination Criterion Satisfied?
Yes
Designate Result
End
No
Evaluate fitness of each individual in population
i := 0
i = M?
Yes
Gen := Gen + 1
No
Select Genetic Operation Probabilistically
P_r
P_c
Select One Individual Based on Fitness
Perform Reproduction
Copy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

Fig. 4. *Flowchart for genetic programming*

- 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 handle correctly certain subcases of the problem (case-splitting);
- the creating role of crossover in recombining valuable parts of more fit parents;
- 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;
- that it is not necessary to determine in advance the size and shape of the 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 $N = k + 2^k$.

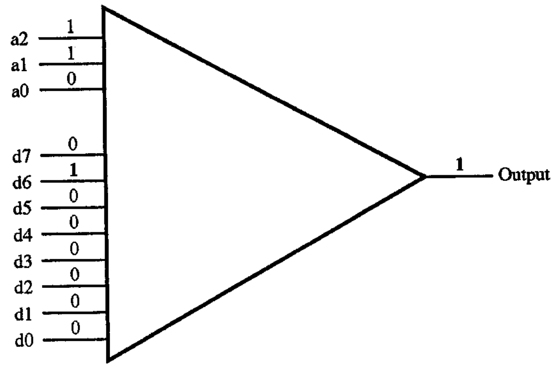

Fig. 5. *Boolean 11-multiplexer*

That is, the input consists of the $k + 2^k$ bits

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

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 5 shows a Boolean 11-multiplexer with an input of 11001000000 and the corresponding output of 1.

There are five major steps in preparing to use genetic programming, namely determining:

1. the set of terminals;
2. the set of primitive functions;
3. the fitness measure;
4. the parameters for controlling the run;
5. the method for designating a result and the criterion for terminating a run.

The first major step in preparing to use genetic programming is the identification of 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 straightforward for this problem. The terminal set for this problem consists of the 11 inputs to the Boolean 11-multiplexer. Thus, the terminal set $\mathcal{T}$ for this problem consists of

$$\mathcal{T} = \{\text{A0}, \text{A1}, \text{A2}, \text{D0}, \text{D1}, \ldots, \text{D7}\}.$$

The second major step in preparing to use genetic programming is the identification of a sufficient set of primitive functions that will be available for constructing the computer programs (S-expressions) that solve the problem. Thus, the function set $\mathcal{F}$ for this problem is

$$\mathcal{F} = \{\text{AND}, \text{OR}, \text{NOT}, \text{IF}\}$$

taking 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). The above function set $\mathcal{F}$ is known to be sufficient to realize any Boolean function.

Since genetic programming operates on an initial population of randomly generated compositions of the available functions and terminals (and later performs genetic operations, such as crossover, on these individuals), each primitive function in the function set should be well defined for any combination of arguments from the range of values returned by every primitive function that it may encounter and the value of every terminal that it may encounter. The above function set $\mathcal{F}$ of primitive functions satisfies the closure property.

The search space for this problem is the set of all LISP S-expressions that can be recursively composed of the primitive functions from the function set $\mathcal{F}$ and terminals from the terminal set $\mathcal{T}$. 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}.

The third major step in preparing to use genetic programming is the identification of the fitness measure for evaluating the goodness of an individual S-expression in the population. Fitness is often evaluated over a number of fitness cases—just as computer programs are typically debugged by examining their output over a number of test 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 (although we could, of course, 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.

It is useful to define a fitness measure called standardized fitness where a smaller value is better and a zero value is best. Since a bigger value of raw fitness is better for this problem, 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 (errors) 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 genetic programming is selecting the values of certain parameters. The two major parameters that are used to control the process are the population size M and the maximum number of generations N_{gen} to be run. N_{gen} is 51 throughout this article. Our choice of 4000 as the population size for this problem reflects an estimate on our part as to the likely complexity of this problem and the practical limitations of available computer memory.

In addition, genetic programming is controlled by a number of additional secondary parameters. Our choice of values for the various secondary parameters that control the runs of genetic programming are the same default values as we have used on numerous other problems (Koza 1992*a*). Specifically, each new generation is created from the preceding generation by applying the fitness proportionate reproduction operation to 10% of the population and by applying the crossover operation to 90% of the population (with both parents selected with a probability proportionate to fitness). In selecting crossover points, 90% were internal (function) points of the tree and 10% were external (terminal) points of the tree. For the practical reason of avoiding the expenditure of large amounts of computer time on an occasional oversized program, the depth of initial random programs was limited to 6 and the depth of programs created by crossover was limited to 17. The individuals in the initial random generation were generated so as to obtain a wide variety of different sizes and shapes among the S-expressions. Fitness is 'adjusted' to emphasize small differences near zero. Spousal selection was also fitness proportionate. Details of the selection of these secondary parameters can be found in Koza (1992*a*). We believe that sufficient information is provided herein and in Koza (1992*a*) to allow replication of the experimental results reported herein, within the limits inherent in a probabilistic algorithm. Common LISP software for genetic programming is listed in Koza (1992*a*).

Finally, the fifth major step in preparing to use genetic programming is the selection of the criterion for terminating a run and the selection of the method for designating a result. 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 article, we will terminate a given run after 51 generations and we designate the best single individual in the population at the time of termination as the result of genetic programming.

We now illustrate genetic programming 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).

None the less, even in this highly unfit initial random population, some individuals are somewhat more fit than others. For this particular run, the individuals in the initial random population had values of standardized fitness ranging from 768 mismatches (i.e. 1280 matches) 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 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 none the less does some things right. For example, it uses 1 of the 3 address bits (A0) as the basis for its action. It could easily have done this wrong and used 1 of the 8 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 1 of the 3 odd-numbered data bits (D1) as its output. Moreover, if A0 is NIL, this individual selects 1 of the 3 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.

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 6 shows the hits histogram of the population for generation 0 of this run of this problem.

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.

The initial random generation is an exercise in blind random search. In going from generation 0 to generation

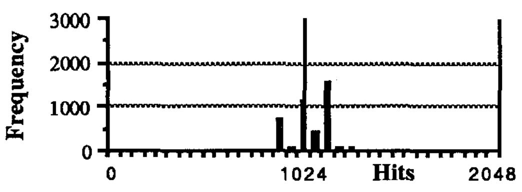

Fig. 6. *Hits histogram for generation 0*

1, genetic programming works with the inevitable genetic variation existing in an initial random population. 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. In this run, 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 7 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.

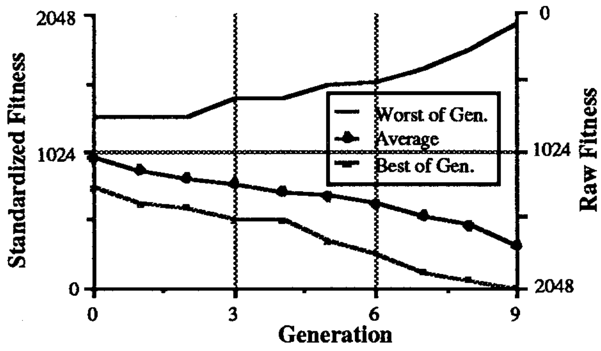

Fig. 7. *Standardized fitness of worst-of-generation individual, average standardized fitness of population, and standardized fitness of best-of-generation individual for generations 0 through 9*

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 1 of the 3 address bits (A0). Second, this best individual from generation 1 incorporates 3 of the 8 data bits as its output, whereas the best individual in generation 0 incorporated only 2 of the 8 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. Note that these 21 individuals are not just copies of the best-of-generation individual from generation 1. Instead, they represent a number of different programs with the same fitness, but different structure and behaviour.

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 centre 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 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 5 of the 8 data bits. By generation 4, the high point of the histogram has moved to 1408 with 1559 individuals. In generation 6, 4 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, and 1672 individuals share this value. Moreover, an additional 887 individuals score 1792.

In generation 9, one individual emerges with a 100% 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 8 shows the 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 genetic programming. 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.

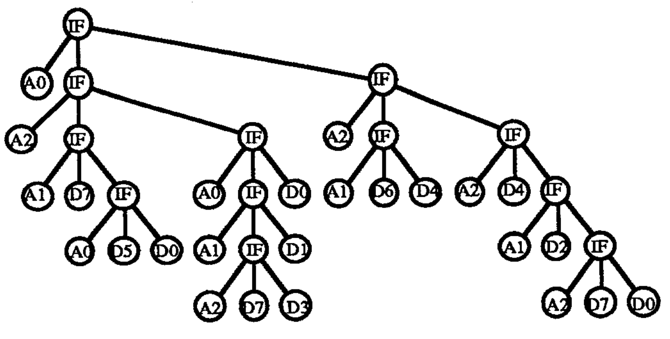

Fig. 8. *100% correct individual from generation 9*

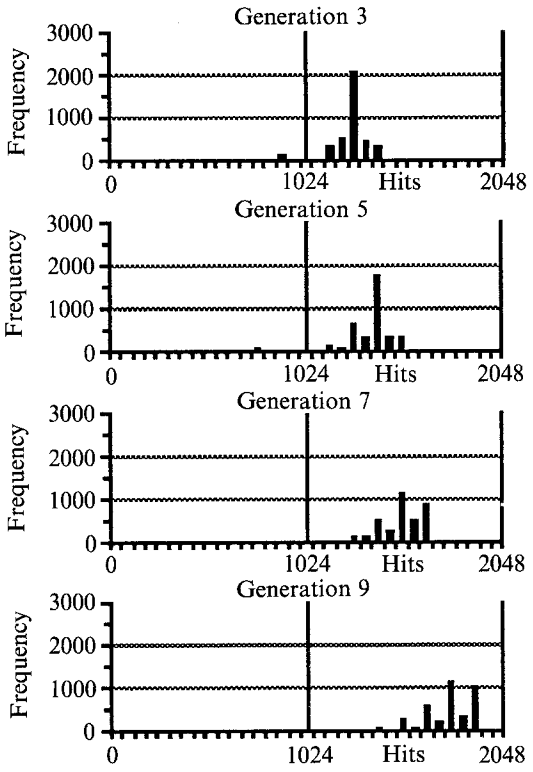

Fig. 9. *Hits histograms for generations 3, 5, 7, and 9 for the 11-multiplexer*

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 1 of the 8 possible data bits.

Figure 9 shows 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 centre 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.

Further insight can be gained by studying the genealogical audit trail consisting of a complete record of the details of each genetic operation that is performed at each generation. The creative role of crossover and case-splitting is illustrated by an examination of the genealogical audit trail for the 100% correct individual emerging at generation 9.

The 100% correct individual emerging at generation 9 is the child resulting from the most common genetic operation

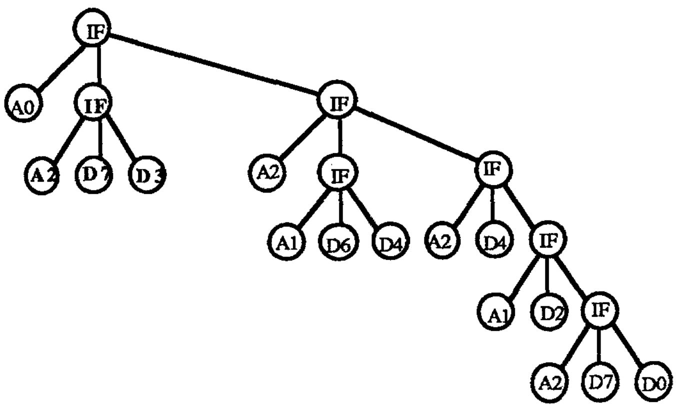

Fig. 10. *First parent (scoring 1792 hits) from generation 8 for 100% correct individual in generation 9*

used in the process, namely crossover. The first parent from generation 8 had rank location of 58 in the population (with a rank of 0 being the very best) 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 10 shows this first parent from generation 8. Note that this first parent starts by examining address bit A0. If A0 is T, the underlined portion then examines address bit A2. It then, partially blindly, makes the output equal D7 or D3 without even considering address bit A1. Moreover, the underlined portion of this individual does not even contain data bits D1 and D5.

On the other hand, when A0 is NIL, this first parent is 100% correct. In that event, it examines A2 and, if A2 is T, it then examines A1 and makes the output equal to D6 or D4 according to whether A1 is T or NIL. Moreover, if A2 is NIL, it 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 and 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 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 11 shows the second parent from generation 8.

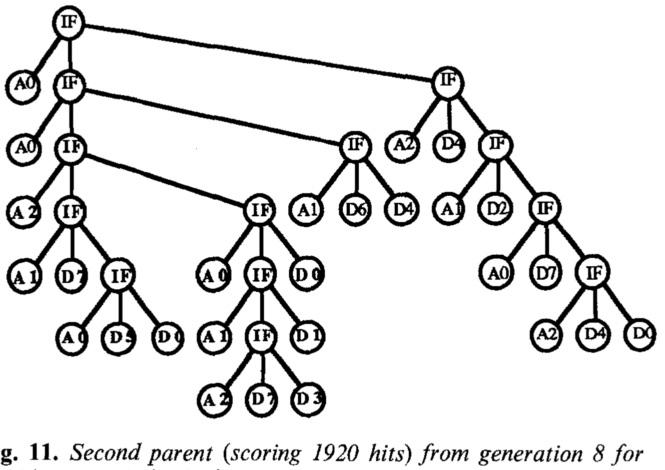

Fig. 11. *Second parent (scoring 1920 hits) from generation 8 for 100% correct individual in generation 9*

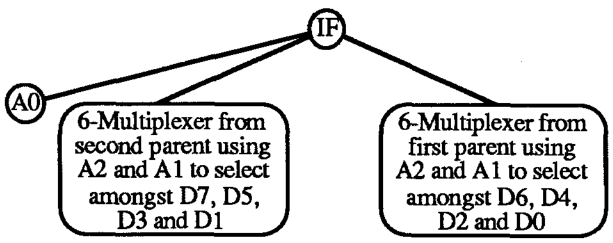
IF
A0
6-Multiplexer from second parent using A2 and A1 to select amongst D7, D5, D3 and D1
6-Multiplexer from first parent using A2 and A1 to select amongst D6, D4, D2 and D0

Fig. 12. *Simplified 100% correct individual from generation 9 shown as a hierarchy of two 6-multiplexers*

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 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, to D5 when the address bits are 101, to D3 when the address bits are 011; and 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 the 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 very well when A0 is NIL (i.e. the even-numbered data bits). 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 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.

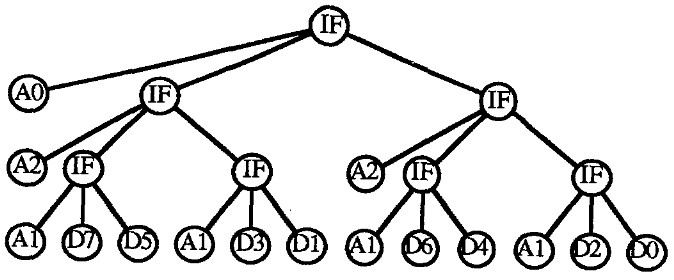

Fig. 13. *Simplified 100% correct individual from generation 9 shown as a hierarchy of two 6-multiplexers*

Figure 12 shows this case splitting by showing the 100% correct offspring from generation 9 as two 6-multiplexers: Fig. 13 also shows this simplified version of the 100% correct individual from generation 9.

Of course, not all crossovers between individuals are useful and productive. In fact, a large number 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 make 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.

The solution to the 11-multiplexer problem in this run was a hierarchy consisting of two 6-multiplexers. In a run where we applied genetic programming to the simpler Boolean 6-multiplexer, we obtained the following 100% correct solution:

(IF (AND A0 A1) D3 (IF A0 D1 (IF A1 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 genetic programming. 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 A1 D2 D0))

three quarters of the time. However, in the specific exceptional fitness case where both address bits (A0 and A1) are both T, the output is the data bit D3.

Default hierarchies are considered desirable in induction problems (Holland, 1986; Holland *et al.*, 1986, Wilson, 1988) because they are often parsimonious and they are a human-like way of dealing with situations.

5. Symbolic regression–empirical data

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 learning of the Boolean multiplexer function is an example 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 conventional 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.

Of course, it is left to the researcher to decide whether to do a linear regression, a quadratic regression, or a higher-order polynomial regression, or whether to try to fit the data points to some non-polynomial family of functions (e.g. sines and cosines of various periodicities, etc.). But, often, *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, in symbolic form, a function that fits a given finite sample of data, by the name *symbolic regression.* It is 'data to function' regression.

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 the relationship

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

from the actual observed noisy time series data. Moreover, suppose that certain additional economic data are also available which are irrelevant to this relationship, but not pre-identified as being irrelevant. 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. Specifically, the 'correct' exchange equation for the United States in the postwar period is the non-linear relationship

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

where 1.6527 is the actual long-term historic postwar value of the M2 velocity of money in the United States (Hallman *et al.* 1989). Interest rates are not a relevant variable in this well-known relationship.

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

- Inflation or price level P (the dependent variable here) is represented by the Gross National Product Deflator (normalized to 1.0) for 1982 (conventionally called GD).
- The gross national product of the economy Q (one of the independent variables) is represented by the annual rate for the United States Gross National Product in billions of 1982 dollars (conventionally called GNP82).
- The money supply M (another of the independent variables) is represented by the monthly values of the seasonally adjusted money stock M2 in billions of dollars, averaged for each quarter (conventionally called M2).
- Interest rates (an independent variable that happens to be irrelevant to the calculation here) are represented by the monthly interest rate yields of 3-month Treasury bills, averaged for each quarter (conventionally called FYGM3).

The four time series used here were obtained from the CITIBASE data base of machine-readable econometric time series (Citibank, 1989).

As a point of reference, 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) is very small, namely 0.077193. The correlation R^2 was 0.993320.

These 120 combinations of the above three independent variables (M2), and the associated value of the dependent variables (GD, GNP82, and FYGM3) are the set from which we will draw 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 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 space of possible S-expressions that can be composed from a set of available functions and arguments.

The appearance of numeric constants (such as the constant 1.6527 in the above correct equation) is typical of relations among empirical data from the real world. Thus, we must deal with the problem of discovering coefficients and constant values while doing symbolic regression.

Constants can be created in genetic programming by adding an ephemeral random constant $\mathscr{R}$ to the terminal set. During the creation of generation 0, whenever the ephemeral random constant $\mathscr{R}$ is chosen for an endpoint of the tree, a random number of an appropriate 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 -1.000 and $+1.000$.

This random generation is done anew each time when an ephemeral $\mathscr{R}$ 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. However, after the initial random generation, the numerous different random constants will be moved around from tree to tree by the crossover operation. In many instances, these constants will be combined via the arithmetic operations in the function set of the problem.

This 'moving around' and 'combining' 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 will usually fit better 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 of the original numerical constants. It is thus possible to genetically evolve numeric constants as required to perform a required symbolic regression on numeric 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 genetic programming is to identify the set of terminals. The terminal set for this problem is

$$\mathrm{T} = \{\mathrm{GNP82}, \mathrm{FM2}, \mathrm{FYGM3}, \mathscr{R}\}.$$

The terminals GNP82, FM2, and FGYM3 correspond to the independent variables of the model and provide access to the values of the time series. In effect, these terminals are functions of the unstated, implicit time variable which ranges over the various quarters.

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

$$\mathrm{F} = \{+, -, *, \%, \mathrm{EXP}, \mathrm{RLOG}\}$$

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

It is necessary to ensure closure by protecting against the possibility of division by zero and the possibility of creating extremely large or small floating-point values. Accordingly, the protected division function % ordinarily returns the quotient; however, if division by zero is attempted, it returns 1.0. The one-argument exponential function EXP ordinarily returns the result of raising e to the power indicated by its one argument. If the result of evaluating EXP or any of the four arithmetic functions would be greater than 10^{10} or less than 10^{-10}, then the nominal value 10^{10} or 10^{-10}, respectively, is returned. The protected logarithm function RLOG returns 0 for an argument of 0 and otherwise returns the logarithm of the absolute value of the argument.

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, polynomial, exponential, logarithmic, non-linear, or otherwise. The unknown functional relationship could be any combination of the functions in the function set. Notice also that we are also not given the known constant value V for the velocity of money. And, notice that we are not told that the 3-month Treasury bill yields (FYGM3) contained in the terminal set and the addition, subtraction, exponential, and logarithm functions are all irrelevant to finding the econometric model for the dependent variable GD of this problem.

The third major step in using genetic programming 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. Population size was 500 here.

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 only 0.009272 over the in-sample period. Figure 14 graphically depicts this best-of-run individual.

This best-of-run individual is equivalent to

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

Notice the sub-tree (* −0.402 0 −0.583) on the left of this best-of-run individual. This sub-expression evaluates to

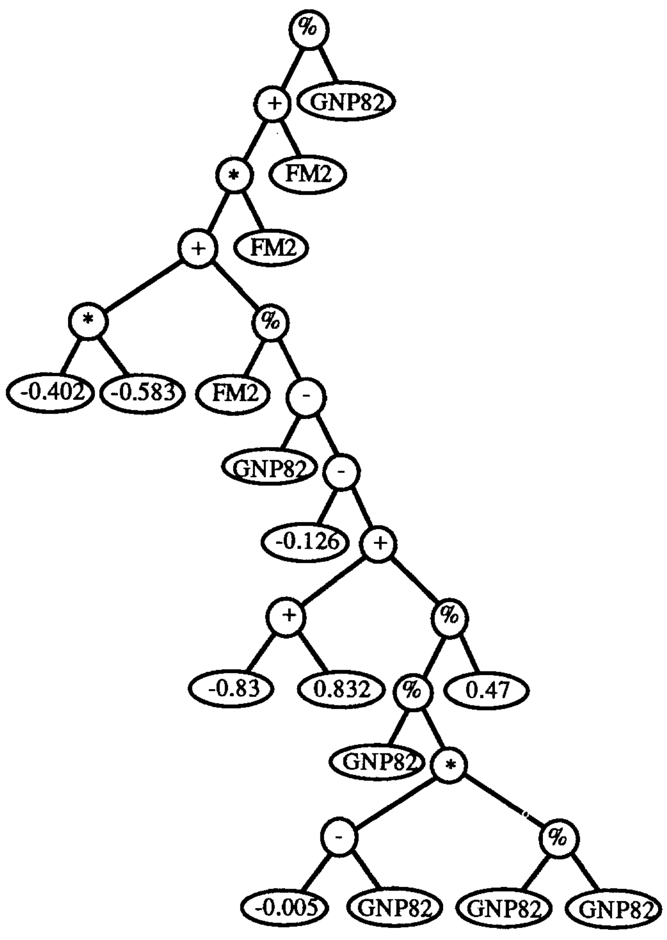

Fig. 14. *Best-of-run individual for exchange equation problem*

+0.234. The numeric constants −0.402 0 and −0.583 were created in generation 0 by the constant creation process. These two constants are combined into a new constant (+0.234), which, in conjunction with other such constants, eventually produces the overall 1.634 constant as the velocity of money.

Although genetic programming has succeeded in finding an expression that fits the given data rather well, there is always a concern that a fitting technique may be overfitting (i.e. memorizing) the data. If a fitting technique overfits the data, the model produced has no ability to generalize to new combinations of the independent variables and therefore has little or no predictive or explanatory value. We can validate the model produced from the 80-quarter in-sample period with the data from the 40-quarter out-of-sample period.

Table 1. *Comparison of in-sample and out-of-sample periods*

Data range	1–120	1–80	81–120
R^2	0.993 480	0.997 949	0.990 614
Sum of squared error	0.075 388	0.009 272	0.066 116

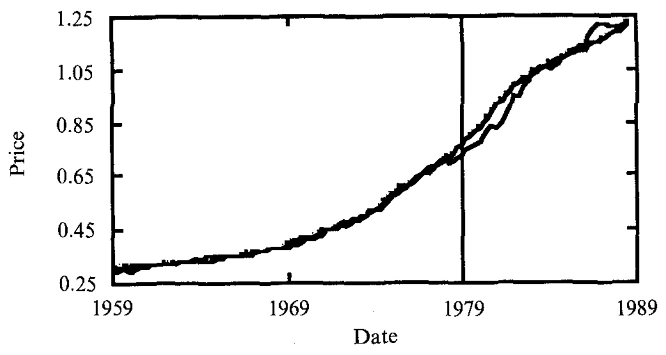

Fig. 15. *Gross national product deflator and fitted series computed from genetically produced model*

Table 1 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.

Figure 15 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 an ordinary line.

Figure 16 shows the residuals from the fitted GD series calculated from the above genetically produced model for 1959:1 to 1988:4.

We can further increase confidence that this genetically evolved model is not overfitting the data by dividing the same 30-year period into a different set of in-sample and out-of-sample periods. When 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, we obtain a virtually identical model. See Koza (1992*a*).

6. Hierarchical automatic function definition–11-parity function

A key goal in machine learning and artificial intelligence is to facilitate the solution of a problem by automatically and dynamically decomposing the problem into simpler subproblems.

A human programmer writing a computer program to solve a problem often creates a subroutine (procedure, function) enabling a common calculation to be performed without tediously rewriting the code for that calculation. For example, a programmer who needed to write a program for Boolean parity functions of several different high orders might find it convenient first to write a subroutine for some lower-order parity function. The code for this low-order parity function would be called at different places and with different combinations of arguments from the main program and the results then combined in the main program to produce the desired higher-order parity function. Specifically, a programmer using the LISP programming language might first write a function definition for the odd-2-parity function xor (exclusive-or) as follows:

```
(defun xor (arg0 arg1)
  (values (or (and arg0 (not arg1))
    (and (not arg0) arg1)))).
```

This function definition (called a 'defun' in LISP) does four things. First, it assigns a name, xor, to the function being defined thereby permitting subsequent reference to it. Second, it identifies the argument list of the function being defined, namely the list (arg0 arg1) containing two dummy variables (formal parameters) called arg0 and arg1. Third, it contains a body which performs the work of the function. Fourth, it identifies the value to be returned by the function. In this example, the single value to be returned is emphasized via an explicit invocation of the 'values' function. This particular function definition has two dummy arguments, returns only a single value, has no side effects, and refers only to the two local dummy variables (i.e. it does not refer to any of the actual variables of the overall problem contained in the 'main' program). However, in general, defined functions may have any number of arguments (including no arguments), may return multiple values (or no values), may or may not perform side effects, and may or may not explicitly refer to the actual (global) variables of the main program.

Once the function xor is defined, it may then be repeatedly called with different instantiations of its arguments from more than one place in the main program. For example, a programmer who needed the even-4-parity at some point in the main program might write

```
(xor (xor d0 d1) (not (xor d2 d3))).
```

Function definitions exploit the underlying regularities and symmetries of a problem by obviating the need to rewrite lines of essentially similar code. A function

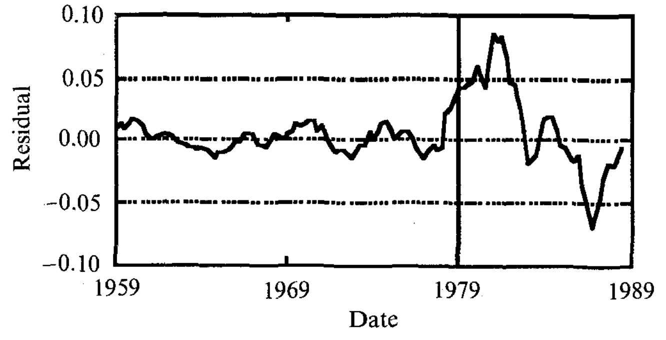

Fig. 16. *Residuals between the gross national product deflator and fitted series computed from genetically produced model*

definition is especially efficient when it is repeatedly called with different instantiations of its arguments. However, the importance of function definition goes well beyond efficiency. The process of defining and calling a function, in effect, decomposes the problem into a hierarchy of subproblems.

The ability to extract a reusable subroutine is potentially very useful in many domains. Consider the problem of discovery of a neural network to recognize patterns presented as an array of pixels. Suppose the solution of a pattern recognition problem requires discovery of a particular feature (e.g. a line end) within the 3×3 pixel region in the upper left corner of an 8×8 array of pixels and also requires discovery of that same feature within a 3×3 pixel region in the lower left corner of the overall array. Existing neural net paradigms can successfully discover the useful feature among the nine pixels p_{11}, p_{12}, p_{13}, p_{21}, p_{22}, p_{23}, p_{31}, p_{32}, p_{33} in the upper left corner of an 8×8 array of pixels and can independently rediscover the same useful feature among the nine pixels p_{61}, p_{62}, p_{63}, p_{16}, p_{71}, p_{72}, p_{73}, p_{81}, p_{82}, p_{83} in the lower left corner of the overall array. But existing neural net paradigms do not provide a way to discover the common feature *just once*, to generalize the feature so that it is not rigidly expressed in terms of particular pixels but is parametrized by its position, and then to reuse the generalized feature detector to recognize occurrences of the feature in different 3×3 pixel regions within the array. That is, existing paradigms do not provide a way to discover a function of nine dummy variables *just once* and to call that function twice (once with $p_{11}, \ldots, p_{33}$ as arguments and once with $p_{61}, \ldots, p_{83}$ as arguments). Such an ability would amount to discovering a nine-input subassembly of neurons with appropriate weights, making a copy of the entire subassembly, implanting the copy elsewhere in the overall neural net, and then connecting nine different pixels as inputs to the subassembly in its new location in the overall neural net.

Hierarchical automatic function definition can be implemented within the context of genetic programming by establishing a constrained syntactic structure for the individual S-expressions in the population (Koza, 1992*a*). Each individual S-expression in the population contains one (or more) function-defining branches and one (or more) 'main' result-producing branches. The result-producing branch may call the defined functions. One defined function may hierarchically refer to another already-defined function (and potentially even itself), although such hierarchical or recursive references will not be used in this article.

6.1. *Learning the even-parity function without hierarchical automatic function definition*

In order to establish the facilitating benefits of hierarchical automatic function definition in genetic programming, we first solve some benchmark problems without using hierarchical automatic function definition.

The Boolean even-parity function of k Boolean arguments returns T (true) if an even number of its arguments are T, and otherwise returns NIL (false).

In applying genetic programming to the even-parity function of k arguments, the terminal set T consists of the k Boolean arguments D0, D1, D2, ... involved in the problem, so that

$$T = \{D0, D1, D2, ...\}.$$

The function set F for all the examples herein consists of the following computationally complete set of four two-argument primitive Boolean functions:

$$F = \{AND, OR, NAND, NOR\}.$$

The Boolean even-parity functions appear to be the most difficult Boolean functions to find via a blind random generative search of S-expressions using the above function set F and the terminal set T. For example, even though there are only 256 different Boolean functions with three arguments and one output, the Boolean even-3-parity function is so difficult to find via a blind random generative search that we did not encounter it at all after randomly generating 10 000 000 S-expressions using this function set F and terminal set T. In addition, the even-parity function appears to be the most difficult to learn using genetic programming using the function set F and terminal set T above (Koza, 1992*a*).

In applying genetic programming to the problem of learning the Boolean even-parity function of k arguments, the 2^k combinations of the k Boolean arguments constitute an exhaustive set of fitness cases for learning this function. The standardized fitness of an S-expression is the sum, over these 2^k fitness cases, of the Hamming distance (error) between the value returned by the S-expression and the correct value of the Boolean function. Standardized fitness ranges between 0 and 2^k; a value closer to zero is better. The raw fitness is equal to the number of fitness cases for which the S-expression is correct (i.e. 2^k minus standardized fitness); a higher value is better.

We first consider how genetic programming would solve the problems of learning the even-3-parity function (three-argument Boolean rule 105), the even-4-parity function (four-argument Boolean rule 38 505), and the even-5-parity function (five-argument Boolean rule 1 771 476 585). In identifying these k-argument Boolean functions in this way, we are employing a numbering scheme wherein the value of the function for the 2^k combinations of its k Boolean arguments are concatenated into a 2^k-bit binary number and then converted to the equivalent decimal number. For example, the $2^3 = 8$ values of the even-3-parity function are 0, 1, 1, 0, 1, 0, 0, and 1 (going from the fitness case consisting of three true arguments to the fitness case consisting of three false arguments). Since

$01101001_2 = 105_{10}$, the even-3-parity function is referred to as three-argument Boolean rule 105.

The terminal set T for the even-3-parity problem consists of

$$T = \{D0, D1, D2\}.$$

In one run of genetic programming using a population size of 4000 (the value of M used consistently in this section, except as otherwise noted), genetic programming discovered the following S-expression containing 45 points (i.e. 22 functions and 23 terminals) with a perfect value of raw fitness of 8 (out of a possible value of $2^3 = 8$) in generation 5:

```
(AND (OR (OR D0 (NOR D2 D1)) D2) (AND (NAND
  (NOR (NOR D0 D2) (AND (AND D1 D1) D1))
  (NAND (OR (AND D0 D1) D2) D0)) (OR (NAND
  (AND D0 D2) (OR (NOR D0 (OR D2 D0)) D1))
  (NAND (NAND D1 (NAND D0 D1)) D2)))).
```

We then considered the even-4-parity function. In one run, genetic programming discovered a program containing 149 points with a perfect value of raw fitness of 16 (out of $2^4 = 16$) in generation 24.

Figure 17 presents two curves, called the performance curves, relating to the even-3-parity function over a series of runs. The curves are based on 66 runs with a population size M of 4000 and a maximum number of generations to be run G of 51. The rising curve in Fig. 17 shows, by generation, the experimentally observed cumulative probability of success, $P(M,i)$, of solving the problem by generation i (i.e. finding at least one S-expression in the population which produces the correct value for all $2^3 = 8$ fitness cases). As can be seen, the experimentally observed value of the cumulative probability of success, $P(M,i)$, is 91% by generation 9 and 100% by generation 21 over the 66 runs. The second curve in Fig. 17 shows, by generation, the number of individuals that must be processed, $I(M,i,z)$, to yield, with probability z, a solution to the problem by generation i. $I(M,i,z)$ is derived from the experimentally observed values of $P(M,i)$. Specifically, $I(M,i,z)$ is the product of the population size M, the generation number i, and the number of independent runs $R(z)$ necessary to yield a solution to the problem with probability z by generation i. In turn, the number of runs $R(z)$ is given by

$$R(z) = \left\lceil \frac{\log(1-z)}{\log(1-P(M,i))} \right\rceil,$$

where the square brackets indicates the ceiling function for rounding up to the next highest integer. The probability z will be 99% herein.

As can be seen, the $I(M,i,z)$ curve reaches a minimum value at generation 9 (highlighted by the light dotted vertical line). For a value of $P(M,i)$ of 91%, the number of independent runs $R(z)$ necessary to yield a solution to the problem with a 99% probability by generation i is 2. The two summary numbers (i.e. 9 and 80 000) in the oval indicate that if this problem is run through to generation 9 (the initial random generation being counted as generation 0), processing a total of 80 000 individuals (i.e. 4000×10 generations $\times 2$ runs) is sufficient to yield a solution to this problem with 99% probability. This number 80 000 is a measure of the computational effort necessary to yield a solution to this problem with 99% probability.

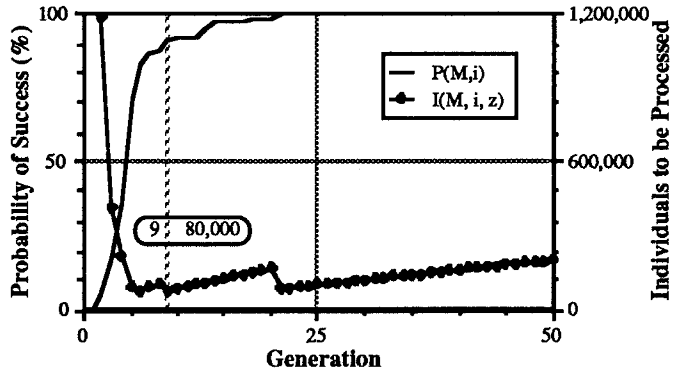

Fig. 17. *Performance curves for even-3-parity function showing that it is sufficient to process 80 000 individuals to yield a solution with 99% probability with genetic programming*

Figure 18 shows similar performance curves for the even-4-parity function based on 60 runs. The experimentally observed cumulative probability of success, $P(M,i)$, is 35% by generation 28 and 45% by generation 50. The $I(M,i,z)$ curve reaches a minimum value at generation 28. For a value of $P(M,i)$ of 35%, the number of runs $R(z)$ is 11. The two numbers in the oval indicate that if this problem is run through to generation 28, processing a total of 1 276 000 (i.e. 4000×29 generations $\times 11$ runs) individuals is sufficient to yield a solution to this problem with 99% probability. Thus, according to this measure of

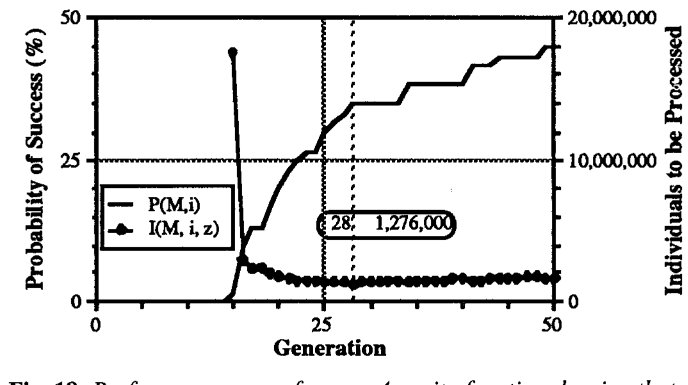

Fig. 18. *Performance curves for even-4-parity function showing that it is sufficient to process 1 276 000 individuals to yield a solution with 99% probability with genetic programming*

computational effort, the even-4-parity problem is about 16 times harder to solve than the even-3-parity problem.

We are unable to extend directly this comparison of the computational effort necessary to solve the even-parity problem with increasing numbers of arguments with our chosen population size of 4000. When the even-5-parity function was run with a population size of 4000 and each run arbitrarily stopped at our chosen maximum number $G = 51$ of generations to be run, no solution was found after 20 runs. (Solutions might well have been found if we had continued the run, but we did not do this.) Even after increasing the population size of 8000 (with $G = 51$), we did not get a solution until our eighth run. This solution contained 347 points.

Notice that the structural complexity (i.e. the total number of function points and terminal points in the S-expression) of the solutions produced in these three cited runs dramatically increased with an increasing number of arguments (i.e. structural complexity was 45, 149, and 347, respectively, above for the 3-, 4-, and 5-parity functions).

The population size of 4000 is undoubtedly not optimal for any particular parity problem and is certainly not optimal for all sizes of parity problems. Nonetheless, it is clear that learning the even-parity functions with increasing numbers of arguments requires dramatically increasing computational effort and that the structural complexity of the solutions become increasingly large.

6.2. *Hierarchical automatic function definition*

The inevitable increase in computational effort and structural complexity for solving parity problems of order greater than 4 could be controlled if we could discover the underlying regularities and symmetries of this problem and then hierarchically decompose the problem into more tractable sub-problems. Specifically, we need to discover a function parametrized by dummy variables that would be helpful in decomposing and solving the problem.

A human programmer writing code for the even-3-parity or even-4-parity functions would probably choose to call upon either the odd-2-parity function (also known as the exclusive- or function XOR) or the even-2-parity function (also known as the equivalence function EQV). For the even-5-parity function and parity functions with additional arguments, our programmer would probably also want to call upon either the even-3-parity (3-argument Boolean rule 105) or the odd-3-parity (3-argument Boolean rule 150). These lower-order parity functions would greatly facilitate writing code for the higher-order parity functions. None of these low-order parity functions is, of course, in our original set F of available primitive Boolean functions.

The potentially helpful role of dynamically evolving useful 'building blocks' in genetic programming has been recognized for some time (Koza, 1990). However, when we talk about 'hierarchical automatic function definition' in this article, we are not contemplating merely defining a function in terms of a sub-expression composed of particular fixed terminals (i.e. actual variables) of the problem. Instead, we are contemplating defining functions *parametrized* by dummy variables (formal parameters). Specifically, if the exclusive-or function XOR were being automatically defined during a run, it would be a version of XOR parametrized by two dummy variables (perhaps called ARG0 and ARG1), not a mere call to XOR with particular fixed actual variables of the problem (e.g. D0 and D1). When this parametrized version of the XOR function is called, its two dummy variables ARG0 and ARG1 would be instantiated with two specific values, which would either be the values of two terminals (i.e. actual variables of the problem) or the values of two expressions (each composed ultimately of terminals). For example, the exclusive-or function XOR might be called via (XOR D0 D1) on one occasion and via (XOR D2 D3) on another occasion. On yet another occasion, XOR might be called via

(XOR (AND D1 D2) (OR D0 D2)),

where the two arguments to XOR are the values returned by the expressions (AND D1 D2) and (OR D0 D2), respectively. Each of these expressions is ultimately composed of the actual variables (i.e. terminals) of the problem.

Moreover, when we talk about 'automatic' and 'dynamic' function definition, the goal is to evolve dynamically a dual structure containing both function-defining branches and result-producing (i.e. value-returning) branches by means of natural selection and genetic operations. We expect that genetic programming will dynamically evolve potentially useful function definitions during the run and also dynamically evolve an appropriate result-producing 'main' program that calls these automatically defined functions.

Note that many existing paradigms for machine learning and artificial intelligence do define functional subunits automatically and dynamically during runs (the specific terminology, of course, being specific to the particular paradigm). For example, when a set of weights is discovered enabling a particular neuron in a neural network to perform some subtask, that learning process can be viewed as a process of defining a function (i.e. a function taking the values of the specific inputs to that neuron as arguments and returning an output signal, perhaps a 0 or 1). Note, however, that the function thus defined can be called only once from only one particular place within the neural network. It is called only in the specific part of the neural net (i.e. the neuron) where it was created and it is called only with the original, fixed set of inputs to that specific neuron. Note also that existing paradigms for neural networks do not provide a way to re-use the set of weights discovered in that part of the network in other parts of the network where a similar subtask must be performed on a different set of inputs. The recent work of Gruau (1992) on recursive solutions to Boolean functions is a notable exception.

Hierarchical automatic function definition can be implemented within the context of genetic programming by establishing a constrained syntactic structure (Koza, 1992*a*, Chapter 19) for the individual S-expressions in the population in which each individual contains one or more function-defining branches and one or more 'main' result-producing branches which may call the defined functions.

The number of result-producing branches is determined by the nature of the problem. Since Boolean parity functions return only a single Boolean value, there would be only one 'main' result-producing branch to the S-expression in the constrained syntactic structure required.

We usually do not know *a priori* the optimal number of functions that will be useful for a given problem or the optimal number of arguments for each such function; however, considerations of computer resources (time, virtual memory usage, CONSing, garbage collection, and memory fragmentation) necessitate that choices be made. Additional computer resources are required for each additional function definition. There is a considerable increase in the computer resources required to support the ever-larger S-expressions associated with each larger number of arguments. There will usually be no advantage to having defined functions that take more arguments than there are terminals in the problem. When Boolean functions are involved, there is no advantage to evolving one-argument function definitions (since the only four one-argument Boolean functions are either in our function set already or constant-valued functions).

Thus, for the Boolean even-4-parity problem, it would seem reasonable to permit one two-argument function definition and one three-argument function definition within each S-expression. Thus, each individual S-expression in the population would have three branches. The first (leftmost) branch permits a two-argument function definition (defining a function called ADF0); the second (middle) branch permits a three-argument function definition (defining a function called ADF1); and the third (rightmost) branch is the result-producing branch. The first two branches are function-defining branches which may or may not be called upon by the result-producing branch.

Figure 19 shows an abstraction of the overall structure of an S-expression with two function-defining branches and one result-producing branch. There are 11 'types' of points in each individual S-expression in the population for this problem. The first eight types are an invariant part of each individual S-expression.

The 11 types are as follows:

1. the root (which will always be the place-holding PROGN function);
2. the top point DEFUN of the function-defining branch for ADF0;
3. the name ADF0 of the function defined by this first function-defining branch;
4. the argument list (ARG0 ARG1) of ADF0;
5. the top point DEFUN of the function-defining branch for ADF1;
6. the name ADF1 of the function defined by this second function-defining branch;
7. the argument list (ARG0 ARG1 ARG2) of ADF1;
8. the top point VALUES of the result-producing branch for the individual S-expression as a whole;
9. the body of ADF0;
10. the body of ADF1;
11. the body of the 'main' result-producing branch.

Syntactic rules of construction govern points of types 9, 10, and 11.

For points of type 9, the body of ADF0 is a composition of functions from the given function set F and terminals from the terminal set A2 of two dummy variables, namely A2 = {ARG0, ARG1}.

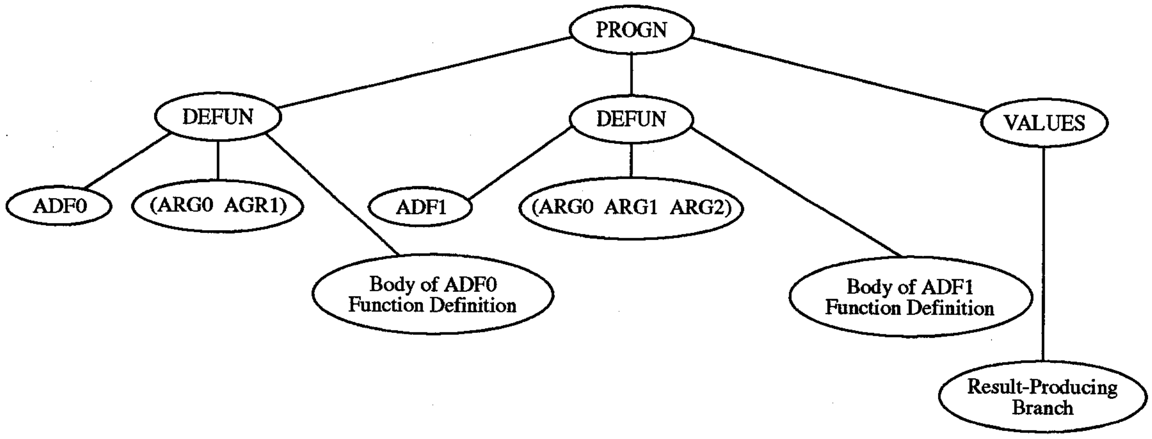

Fig. 19. *Abstraction of the overall structure of an S-expression with two function-defining branches and the one result-producing branch*

For the points of type 10, the body of ADF1 is a composition of functions from the original given function set F *along with* ADF0 and terminals from the set A3 of three dummy variables, namely A3 = {ARG0, ARG1, ARG2}. Thus, the body of ADF1 is capable of calling upon ADF0.

For the points of type 11, the body of the result-producing branch is a composition of terminals (i.e. actual variables of the problem) from the terminal set T, namely T = {D0, D1, D2, D3}, as well as functions from the set F3. F3 contains the four original functions from the function set F as well as the two-argument function ADF0 defined by the first branch and the three-argument function ADF1 defined by the second branch. That is, the function set F3 is

F3 = {AND, OR, NAND, NOR, ADF0, ADF1},

taking two, two, two, two, two, and three arguments, respectively. Thus, the result-producing branch is capable of calling the two defined functions ADF0 and ADF1.

When the overall S-expression in Fig. 19 is evaluated, the PROGN evaluates each branch; however, the value(s) returned by the PROGN consists only of the value(s) returned by the VALUES function in the final result-producing branch.

Note that one might consider including the terminals from the terminal set T (i.e. the actual variables of the problem) in the function-defining branches; however, we do not do so here.

In what follows, genetic programming will be allowed to evolve two function definitions in the function-defining branches of each S-expression and then, at its discretion, to call one, two, or none of these defined functions in the result-producing branch. We do not specify what functions will be defined in the two function-defining branches. We do not specify whether the defined functions will actually be used. As we have already seen it is possible to solve this problem without any function definition by evolving the correct program in the result-producing branch. We do not favour one function-defining branch over the other. We do not require that a function-defining branch use all of its available dummy variables. The structure of all three branches is determined by the combined effect, over many generations, by the selective pressure exerted by the fitness measure and by the effects of the operations of Darwinian fitness proportionate reproduction and crossover.

Since a constrained syntactic structure is involved, we must create the initial random generation so that every individual S-expression in the population has the syntactic structure specified by the syntactic rules of construction presented above. Specifically, every individual S-expression must have the invariant structure represented by the eight points of types 1 through 8. Specifically, the bodies of ADF0 (type 9), ADF1 (type 10), and the result-producing branch (type 11) must be composed of the functions and terminals specified by the above syntactic rules of construction.

Moreover, since a constrained syntactic structure is involved, we must perform structure-preserving crossover so as to ensure the syntactic validity of all offspring as the run proceeds from generation to generation. Structure-preserving crossover is implemented by first allowing the selection of the crossover point in the first parent to be any point from the body of ADF0 (type 9), ADF1 (type 10), or the result-producing branch (type 11). However, once the crossover point in the first parent has been selected, the crossover point of the second parent must be of the same type (i.e. types 9, 10, or 11). This restriction on the selection of the crossover point of the second parent assures syntactic validity of the offspring.

6.3. *Even-4-parity function*

Each S-expression in the population for solving the even-4-parity function has one result-producing branch and two function-defining branches, each permitting the definition of one function of three dummy variables.

In one run of the even-4-parity function, the following 100%-correct solution containing 45 points (not counting the invariant points of types 1 through 8) with a perfect value of 16 for raw fitness appeared on generation 4:

```
(PROGN (DEFUN ADF0 (ARG0 ARG1 ARG2)
            (NOR (NOR ARG2 ARG0)
            (AND ARG0 ARG2)))
   (DEFUN ADF1 (ARG0 ARG1 ARG2)
     (NAND (ADF0 ARG2 ARG2 ARG0)
       (NAND (ADF0 ARG2 ARG1 ARG2)
         (ADF0 (OR ARG2 ARG1)
           (NOR ARG0 ARG1)
           (ADF0 ARG1 ARG0
             ARG2)))))
   (VALUES
     (ADF0 (ADF1 D1 D3 D0)
       (NOR (OR D2 D3) (AND D3 D3))
       (ADF0 D3 D3 D2)))).
```

The first branch of this best-of-run S-expression is a function definition establishing the defined function ADF0 as the two-argument exclusive-or (XOR) function. The definition of ADF0 ignores one of the available dummy variables, namely ARG1. The second branch of the S-expression calls upon the defined function ADF0 (i.e. XOR) to define ADF1. This second branch appears

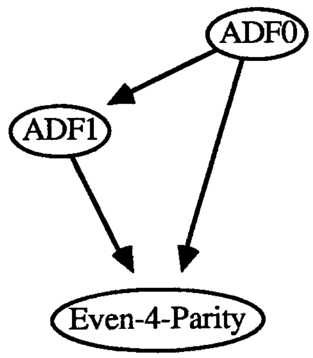

Fig. 20. *Hierarchy (lattice) of function definitions*

to use all three available dummy variables; however, it reduces to the two-argument equivalence function EQV. The result-producing (i.e. third) branch of this S-expression uses all four terminals and both ADF0 and ADF1 to solve the even-4-parity problem. This branch reduces to

(ADF0 (ADF1 D1 D0) (ADF0 D3 D2)).

which is equivalent to

(XOR (EQV D1 D0) (XOR D3 D2)).

That is, genetic programming decomposed the even-4-parity problem into two different parity problems of lower order (i.e. XOR and EQV).

Figure 20 shows the hierarchy (lattice) of function definitions used in this solution to the even-4-parity problem. Note also that the second of the two functions in this decomposition (i.e. EQV) was defined in terms of the first (i.e. XOR).

Note that we did not specify that the exclusive-or XOR function would be defined in ADF0, as opposed to, say, the equivalence function, the if-then function, or any other Boolean function. Similarly, we did not specify what would be evolved in ADF1. Genetic programming created the two-argument defined functions ADF0 and ADF1 on its own to help solve this problem. Having done this, genetic programming then used ADF0 and ADF1 in an appropriate way in the result-producing branch to solve the problem. Notice that the 45 points above are considerably fewer than the 149 points contained in the S-expression cited earlier for the even-4-parity problem.

Figure 21 presents the performance curves based on 23 runs for the even-4-parity with hierarchical automatic function definition. The cumulative probability of success $P(M, i)$ is 91% by generation 10 and 100% by generation 50. The two numbers in the oval indicate that if this problem is run through to generation 10, processing a total of 88 000 individuals (i.e. 4000 × 11 generations × 2 runs) is sufficient to yield a solution to this problem with 99% probability.

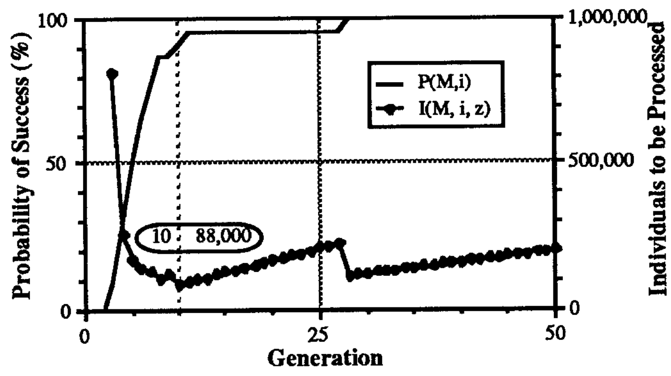

Fig. 21. *Performance curves for the even-4-parity problem show that it is sufficient to process 88 000 individuals to yield a solution with hierarchical automatic function definition*

6.4. *Even 5-parity function*

Each program in the population for solving the even-5-parity function (and all higher-order parity functions herein) has one result-producing branch and two function-defining branches, each permitting the definition of one function of four dummy variables.

In one run of the even-5-parity problem, the 100%-correct solution contains 160 points and emerged on generation 12. The first branch is equivalent to the four-argument Boolean rule 50 115 which is an even-2-parity function that ignores two of the four available dummy variables. The second branch is equivalent to the four-argument Boolean rule 38 250, which is equivalent to

(OR (AND (NOT ARG2) (XOR ARG3 ARG0))
(AND ARG2 (XOR ARG3 (XOR ARG1
ARG0)))).

Notice that this rule is not a parity function of any kind. The result-producing (i.e. third) branch calls on defined functions ADF0 and ADF1 and solves the problem.

The even 5-parity problem can be similarly solved with 99% probability with genetic programming using hierarchical automatic function definition by processing a total of 144 000 individuals.

6.5. *Parity functions with 7 to 10 arguments*

The even 6-, and 7-parity problems can be similarly solved with 99% probability with genetic programming using hierarchical automatic function definition by processing a total of 864 000, and 1 440 000 individuals, respectively.

The 8-, 9-, and 10-parity problems can be similarly solved using hierarchical automatic function definition. Each problem was solved within the first four runs. We did not perform sufficient additional runs to compute a performance curve for these higher-order parity problems.

6.6. *Even-11-parity function*

In one run of the even-11-parity function, the following best-of-generation individual containing 220 points and attaining a perfect value of raw fitness of 2048 appeared in generation 21:

```
(PROGN (DEFUN ADF0 (ARG0 ARG1 ARG2 ARG3)
          (NAND (NOR (NAND (OR ARG2 ARG1)
          (NAND ARG1 ARG2)) (NOR (OR ARG1
          ARG0) (NAND ARG3 ARG1))) (NAND
          (NAND (NAND (NAND ARG1 ARG2)
          ARG1) (OR ARG3 ARG2)) (NOR (NAND
          ARG2 ARG3) (OR ARG1 ARG3)))))
       (DEFUN ADF1 (ARG0 ARG1 ARG2 ARG3)
          (ADF0 (NAND (OR ARG3 (OR ARG0
          ARG0)) (AND (NOR ARG1 ARG1)
          (ADF0 ARG1 ARG1 ARG3 ARG3)))
          (NAND (NAND (ADF0 ARG2 ARG1
          ARG0 ARG3) (ADF0 ARG2 ARG3 ARG3
          ARG2)) (ADF0 (NAND ARG3 ARG0)
          (NOR ARG0 ARG1) (AND ARG3 ARG3)
          (NAND ARG3 ARG0))) (ADF0 (NAND
          (OR ARG0 ARG0) (ADF0 ARG3 ARG1
          ARG2 ARG0)) (ADF0 (NOR ARG0
          ARG0) (NAND ARG0 ARG3) (OR ARG3
          ARG2) (ADF0 ARG1 ARG3 ARG0
          ARG0)) (NOR (ADF0 ARG2 ARG1 ARG2
          ARG0) (NAND ARG3 ARG3)) (AND
          (AND ARG2 ARG1) (NOR ARG1 ARG2)))
          (AND (NAND (OR ARG3 ARG2) (NAND
          ARG3 ARG3)) (OR (NAND ARG3 ARG3)
          (AND ARG0 ARG0)))))
       (VALUES
          (OR (ADF1 D1 D0 (ADF0 (ADF1 (OR
          (NAND D1 D7) D1) (ADF0 D1 D6 D2 D6)
          (ADF1 D6 D6 D4 D7) (NAND D6 D4))
          (ADF1 (ADF0 D9 D3 D2 D6) (OR D10
          D1) (ADF1 D3 D4 D6 D7) (ADF0 D10 D8
          D9 D5)) (ADF0 (NOR D6 D9) (NAND D1
          D10) (ADF0 D10 D5 D3 D5) (NOR D8
          D2)) (OR D6 (NOR D1 D6))) D1) (NOR
          (NAND D1 D10) (ADF0 (OR (ADF0 D6
          D2 D8 D4) (OR D4 D7)) (NOR D10 D6)
          (NOR D1 D2) (ADF1 D3 D7 D7 D6)))))).
```

The first branch of this S-expression defined the four-argument defined function ADF0 (four-argument Boolean rule 50 115) which ignored two of its four arguments. ADF0 is equivalent to the even-2-parity function, namely

```
(EQV ARG1 ARG2).
```

The second branch defined a four-argument defined function ADF1 which is equivalent to the even-4-parity function. Substituting the definitions of the defined functions ADF0 and ADF1, the result-producing (i.e. third) branch becomes:

```
(OR (EVEN-4-PARITY
      D1
      D0
      (EVEN-2-PARITY
        (EVEN-4-PARITY
        (EVEN-2-PARITY D3 D2)
        (OR D10 D1)
        (EVEN-4-PARITY D3 D4 D6 D7)
        (EVEN-2-PARITY D8 D9))
      (EVEN-2-PARITY (NAND D1 D10)
          (EVEN-2-PARITY D5 D3)))
      D1)
    (NOR (NAND D1 D10)
        (EVEN-2-PARITY (NOR D10 D6)
          (NOR D1 D2))))
```

which is equivalent to the target even-11-parity function. Note that the even-2-parity function (ADF0) appears six times in this solution and that the even-4-parity function (ADF1) appears three times. Note that this entire solution for the even-11-parity function contains only 220 points (compared with 347 points for the solution to the *mere even-5-parity* without hierarchical automatic function definition).

Figure 22 shows the simplified version of the result-producing branch of this best-of-run individual for the even-11-parity problem. As can be seen, the even-11-parity problem was decomposed into a composition of even-2-parity functions and even-4-parity functions.

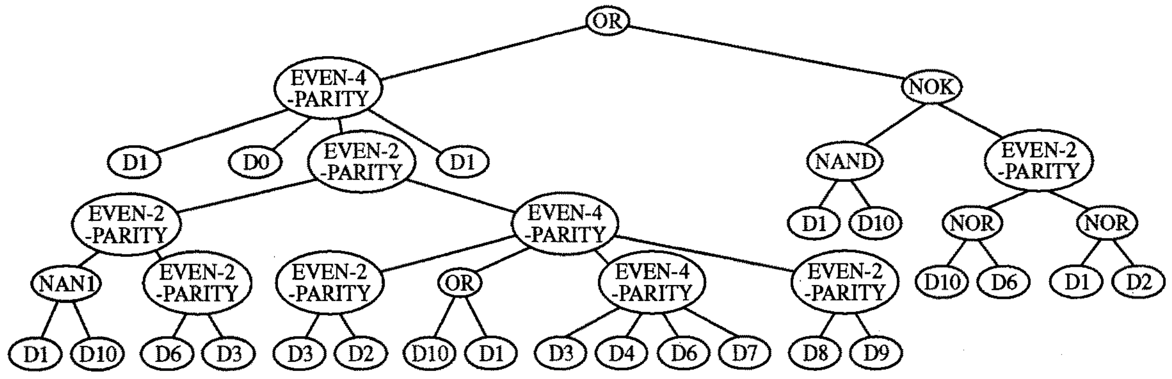

Fig. 22. *The best-of-run individual from generation 21 of one run of the even-11-parity problem is a composition of even-2-parity and even-4-parity functions*

We found the above solution to the even-11-parity problem on our first completed run. The search space of 11-argument Boolean functions returning one value is of size $2^{2048} \approx 10^{616}$. The even-11-parity problem was solved by decomposing into parity functions of lower orders.

6.7. *Summary of hierarchical automatic function definition*

Thus, the problem of learning various higher order even-parity functions can be solved with the technique of hierarchical automatic function definition in the context of genetic programming. Moreover, as can be seen in Table 2, the technique of hierarchical automatic function definition facilitates the solution of these problems. That is, when problems are decomposed into a hierarchy of function definitions and calls, many fewer individuals must be processed in order to yield a solution to the problem. Moreover, the solutions discovered are comparatively smaller in terms of their structural complexity.

Automatic function definition has also been applied to the problem of discovery of impulse response functions (Koza *et al.*, 1993).

7. Additional examples of genetic programming

Genetic programming can be applied in many additional problem domains, including the following:

- evolution of a subsumption architecture for controlling a robot to follow walls or move boxes (Koza, 1992*d*; Koza and Rice, 1992*b*);
- discovering inverse kinematic equations to control the movement of a robot arm to a designated target point;
- emergent behaviour (e.g. discovering a computer program which, when executed by all the ants in an ant colony, enables the ants to locate food, pick it up, carry it to the nest, and drop pheromones along the way so as to recruit other ants into cooperative behaviour);
- symbolic integration, symbolic differentiation, and symbolic solution of general functional equations (including differential equations with initial conditions);
- planning (e.g. navigating an artificial ant along a trail, developing a robotic action sequence that can stack an arbitrary initial configuration of blocks into a specified order);
- generation of high-entropy sequences of random numbers;
- induction of decision trees for classification;
- optimization problems (e.g. finding an optimal food foraging strategy for a lizard);
- sequence induction (e.g. inducing a recursive computational procedure for generating sequences such as the Fibonacci sequence);
- automatic programming of cellular automata;
- finding minimax strategies for games (e.g. differential pursuer–evader games, discrete games in extensive form) by both evolution and co-evolution;
- automatic programming (e.g. discovering a computational procedure for solving pairs of linear equations, solving quadratic equations for complex roots, and discovering trigonometric identities);
- simultaneous architectural design and training of neural networks (Koza and Rice, 1991).

Additional information and examples can be found in Koza (1992*a*).

Table 2. *Number of individuals $I(M, i, z)$ required to be processed to yield a solution to various even-parity problems with 99% probability—with and without hierarchical automatic function definition*

Size of parity function	Without hierarchical automatic function definition	With hierarchical automatic function definition
3	80 000	
4	1 276 000	88 000
5		144 000
6		864 000
7		1 440 000

8. 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. The technique of hierarchical automatic function definition can facilitate the solution of problems.

Acknowledgements

Christopher Jones prepared the figures for the econometric problem. James P. Rice of the Knowledge Systems Laboratory at Stanford University created all the other figures in this article and did all or some of the computer programming on a Texas Instruments Explorer computer for various problems in this article.

References

Belew, R. and Booker, L. (EDS) (1991). *Proceedings of the Fourth International Conference on Genetic Algorithms.* Morgan Kaufmann, San Mateo, CA.

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

Cramer, N. L. (1985). A representation for the adaptive generation of simple sequential programs. In *Proceedings of an International Conference on Genetic Algorithms and Their Applications,* ed. J. Grefenstette. Lawrence Erlbaum, Hillsdale, NJ.

Davidor, Y. (1991). *Genetic Algorithms and Robotics.* World Scientific, Singapore.

Davis, L. (ED) (1987). *Genetic Algorithms and Simulated Annealing.* Pitman, London.

Davis, L. (1991). *Handbook of Genetic Algorithms.* Van Nostrand Reinhold, New York.

Forrest, S. (ED). (1990). *Emergent Computation: Self-Organizing, Collective, and Cooperative Computing Networks.* MIT Press, Cambridge, MA.

Fujiki, C. and Dickinson, J. (1987). Using the genetic algorithm to generate LISP source code to solve the prisoner's dilemma. In *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms,* ed. J. Grefenstette. Lawrence Erlbaum, Hillsdale, NJ.

Goldberg, D. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning,* Addison-Wesley, Reading, MA.

Goldberg, D. E., Korb, B. and Deb, K. (1989). Messy genetic algorithms: motivation, analysis, and first results. *Complex Systems,* **3**, 493–530.

Gruau, F. (1992). Genetic synthesis of Boolean neural networks with a cell rewriting developmental process. In *Proceedings of the Workshop on Combinations of Genetic Algorithms and Neural Networks 1992,* ed. J. D. Schaffer and D. Whitley. The IEEE Computer Society Press.

Hallman, J. J., Porter, R. D. and Small, D. H. (1989). *M2 per Unit of Potential GNP as an Anchor for the Price Level.* Board of Governors of the Federal Reserve System. Staff Study 157, Washington, DC.

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

Holland, J. H. (1986). Escaping brittleness: the possibilities of general-purpose learning algorithms applied to parallel rule-based systems. In *Machine Learning: An Artificial Intelligence Approach, Volume II,* ed. R. S. Michalski *et al.* pp. 593–623. Morgan Kaufmann, Los Altos, CA.

Holland, J. H., Holyoak, K. J., Nisbett, R. E. and Thagard, P. A. (1986). *Induction: Processes of Inference, Learning, and Discovery.* MIT Press, Cambridge, MA.

Koza, J. R. (1990). *Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems.* Stanford University Computer Science Department technical report STAN-CS-90-1314.

Koza, J. R. (1992*a*). *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA.

Koza, J. R. (1992*b*). Genetic programming: genetically breeding populations of computer programs to solve problems. In *Dynamic, Genetic, and Chaotic Programming,* ed. B. Soucek and the IRIS Group. John Wiley, New York.

Koza, J. R. (1992*c*). Hierarchical automatic function definition in genetic programming. In *Proceedings of Workshop on the Foundations of Genetic Algorithms and Classifier Systems, Vail, Colorado 1992,* ed. D. Whitley. Morgan Kaufmann, San Mateo, CA.

Koza, J. R. (1992*d*). Evolution of subsumption using genetic programming. In *Proceedings of European Conference on Artificial Life, Paris, December 1991,* ed. P. Bourgine and F. Varela. MIT Press, Cambridge, MA.

Koza, J. R. and Keane, M. A. (1990*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, pp. 198–201. Lawrence Erlbaum, Hillsdale, NJ.

Koza, R. and Keane, M. A. (1990*b*). 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, France, June, 1990,* pp. 47–56. Springer-Verlag, Berlin.

Koza, J. R. and Rice, J. P. (1991). Genetic generation of both the weights and architecture for a neural network. In *Proceedings of International Joint Conference on Neural Networks, Seattle, July 1991.* Volume II, pp. 397–404. IEEE Press.

Koza, J. R. and Rice, J. P. (1992*a*). *Genetic Programming: The Movie.* MIT Press, Cambridge, MA.

Koza, J. R. and Rice, J. P. (1992*b*). Automatic programming of robots using genetic programming. In *Proceedings of Tenth National Conference on Artificial Intelligence,* pp. 194–201. AAAI Press/MIT Press, Menlo Park, CA.

Koza, J. R., Keane, M. A. and Rice, J. P. (1993). Performance improvement of machine learning via automatic discovery of facilitating functions as applied to a problem of symbolic system identification. In *1993 IEEE International Conference on Neural Networks, San Francisco*, Volume I, pp. 191–198. IEEE Press, Piscataway, NJ.

Langton, C., Taylor, C., Farmer, J. D. and Rasmussen, S. (EDS). (1992). *Artificial Life II, SFI Studies in the Sciences of Complexity*, Volume X. Addison-Wesley, Redwood City, CA.

Meyer, J.-A. and Wilson, S. W. (1991). *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior, Paris. September 24–28, 1990.* MIT Press, Cambridge, MA.

Michaelewicz, Z. (1992). *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, New York.

Rawlins, G. (ED) (1991). *Proceedings of Workshop on the Foundations of Genetic Algorithms and Classifier Systems, Bloomington, Indiana, July 15–18, 1990.* Morgan Kaufmann, San Mateo, CA.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, **3**, 210–229.

Schaffer, J. D. (ED) (1989). *Proceedings of the Third International Conference on Genetic Algorithms*. Morgan Kaufmann, San Mateo, CA.

Schwefel, H.-P. and Maenner, R. (EDS) (1991). *Parallel Problem Solving from Nature*. Springer-Verlag, Berlin.

Smith, S. F. (1980). *A Learning System Based on Genetic Adaptive Algorithms*. PhD dissertation, University of Pittsburgh, Pittsburgh, PA.

Whitley, D. (ED) (1992). *Proceedings of Workshop on the Foundations of Genetic Algorithms and Classifier Systems, Vail, Colorado 1992.* Morgan Kaufmann, San Mateo, CA.

Wilson, S. W. (1987*a*). Classifier systems and the animat problem. *Machine Learning*, **3**, 199–228.

Wilson, S. W. (1987*b*). Hierarchical credit allocation in a classifier system. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pp. 217–220. Morgan Kaufmann, San Mateo, CA.

Wilson, S. W. (1988). Bid competition and specificity reconsidered. *Jounal of Complex Systems*, **2**, 705–723.