Hierarchical Automatic Function Definition in Genetic Programming

John R. Koza

Computer Science Department
Stanford University
Stanford, CA 94305 USA
E-MAIL: Koza@Sunburn.Stanford.Edu
PHONE: 415-941-0336 FAX: 415-941-9430

Abstract

A key goal in machine learning and artificial intelligence is to automatically and dynamically decompose problems into simpler problems in order to facilitate their solution. This paper describes two extensions to genetic programming, called "automatic" function definition and "hierarchical automatic" function definition, wherein functions that might be useful in solving a problem are automatically and dynamically defined during a run in terms of dummy variables. The defined functions are then repeatedly called from the automatically discovered "main" result-producing part of the program with different instantiations of the dummy variables. In the "hierarchical" version of automatic function definition, automatically defined functions may call other automatically defined functions, thus creating a hierarchy of dependencies among the automatically defined functions. The even-11-parity problem was solved using using hierarchical automatic function definition.

1 INTRODUCTION AND OVERVIEW

A key goal in machine learning and artificial intelligence is to automatically and dynamically decompose problems into simpler subproblems in order to facilitate their solution.

When a human programmer writes a computer program to solve a problem, he often creates a sub-routine defined in terms of dummy variables (formal parameters) enabling a common calculation to be performed for different instantiations of the dummy variables. For example, suppose a programmer needed the value of the exponential of X, Y and $3Z^2$ at different points in a main program. In the LISP programming language, the programmer might write a function definition such as

```
(defun our-exp (dv)
  (+ 1.0  dv  (* 0.5 dv dv)  (* 0.1667 dv dv dv)).
```

This function definition (defun) assigns a name (our-exp) to the function being defined; it identifies the list (dv) containing the single dummy variable (formal parameter) dv as the argument list to the function; and it contains a body which performs the work (which here happens to be an approximation to the exponential function consisting of the first four terms of the Taylor series). This particular example of a function definition (and all those in this paper) returns a single numerical value, does not refer to any of the actual variables of the overall problem (i.e., contains only a dummy variable), and has no side effects; however, in general, functions may return multiple values, refer to the actual variables of the overall problem, and perform side effects.

Once the function our-exp is defined, it can then be called an arbitrary number of times with different instantiations of its dummy variable dv.

Function definitions exploit the underlying regularities and symmetries of a problem by obviating the need to tediously rewrite lines of essentially similar code. The process of defining and calling a function, in effect, decomposes the problem into a hierarchy of subproblems.

Genetic programming (Koza 1991, 1992) provides a way to genetically breed a population of computer programs in order to solve a problem. The new processes of "automatic" function definition and "hierarchical automatic" function definition described in this paper provide a way to automatically define and call potentially useful functions on-the-fly during a run of genetic programming.

In section 2 of this paper, we review how the even-parity function of increasing numbers of arguments can be solved with genetic programming. In section 3, we describe the new process of "automatic" function definition and show how it is helpful in facilitating the solution of the Boolean even-parity problem. In section 4, we describe the new process of "hierarchical automatic" function definition and show how it is helpful in facilitating the solution of the even-parity function. Specifically, we use hierarchical automatic function definition to solve the even-parity function for up to eleven arguments. In section 5, we indicate that the technique of hierarchical automatic function definition can be successfully applied to other Boolean functions.

2 LEARNING THE EVEN-PARITY-FUNCTION WITH GENETIC PROGRAMMING

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}.
```

Compositions of functions from the function set F and terminals from the terminal set T are called symbolic expressions (S-expressions) in the LISP programming language. An S-expression can be represented as a rooted, point-labeled tree with ordered branches in which the root and other internal points of the tree are labeled with functions and in which the external points of the tree are labeled with terminals. These trees correspond to the parse trees found in other programming languages.

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 this function set F and terminal set T (Koza 1992).

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.

To establish a baseline for demonstrating the effectiveness of the new processes of automatic function definition and hierarchical automatic function definition, 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).

Throughout this paper, we employ a numbering scheme for identifying a k-argument Boolean function 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 4,000 (the value of *M* used consistently herein, 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 the following S-expression containing 149 points with a perfect value of raw fitness of 16 (out of $2^4 = 16$) in generation 24:

```
(AND (OR (OR (OR (NOR D0 (NOR D2 D1)) (NAND (OR (NOR (AND D3
D0) D2) (NAND D0 (NOR D2 (AND D1 (OR D3 D2))))) D3)) (AND (AND
D1 D2) D0)) (NAND (NAND (NAND D3 (OR (NOR D0 (NOR (OR D3 D2)
D2)) (NAND (AND (AND (AND D3 D2) D3) D2) D3))) (NAND (OR (NAND
(OR D0 (OR D0 D1)) (NAND D0 D1)) D3) (NAND D1 D3))) D3)) (OR
(OR (NOR (NOR (AND (OR (NOR D3 D0) (NOR (NOR D3 (NAND (OR (NAND
D2 D2) D2) D2)) (AND D3 D2))) D1) (AND D3 D0)) (NOR D3 (OR D0
D2))) (NOR D1 (AND (OR (NOR (AND D3 D3) D2) (NAND D0 (NOR D2
(AND D1 D0)))) (OR (OR D0 D3) (NOR D0 (NAND (OR (NAND D2 D2)
D2) D2)))))) (AND (AND D2 (NAND D1 (NAND (AND D3 (NAND D1 D3))
(AND D1 D1)))) (OR D3 (OR D0 (OR D0 D1)))))).
```

Figure 1 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 4,000 and a maximum number of generations to be run *G* of 51 (the value of *G* used consistently herein).

The rising curve in figure 1 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.

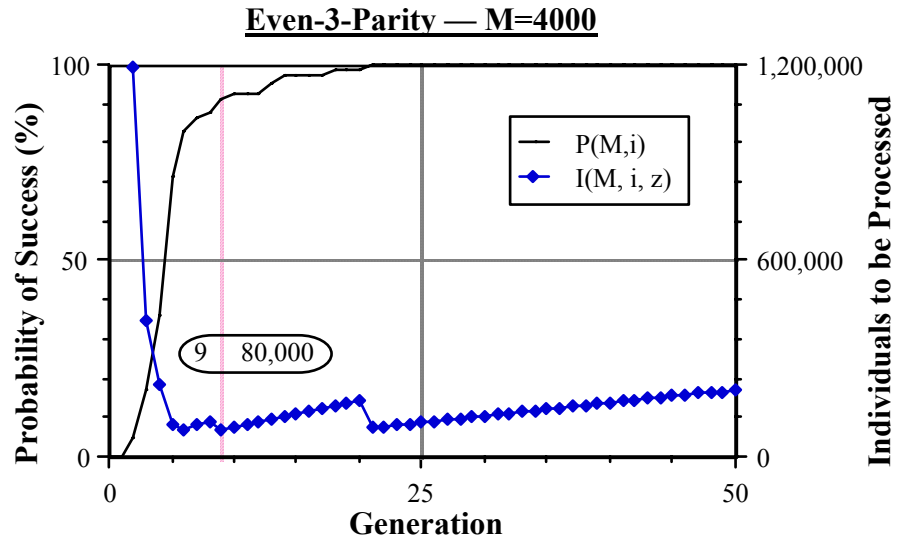

Figure 1 Performance curves for even-3-parity function showing that it is sufficient to process 80,000 individuals to yield a solution with 99% probability with genetic programming.

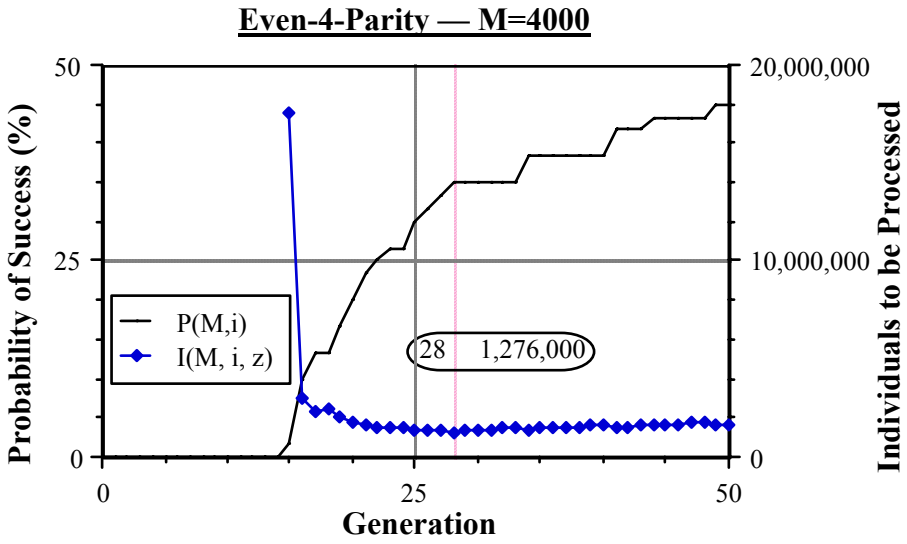

Figure 2 Performance curves for even-4-parity function showing that it is sufficient to process 1,276,000 individuals to yield a solution with 99% probability with genetic programming.

The second curve in figure 1 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. Throughout this paper, the probability z will be 99%.

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., 4,000 ∞ 10 generations ∞ 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.

Figure 2 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., 4,000 ∞ 29 generations ∞ 11 runs) individuals is sufficient to yield a solution to this problem with 99% probability.

Thus, according to this measure of computational effort, the even-4-parity problem is about 16 times harder to solve than the even-3-parity problem.

We are unable to directly extend this comparison of the computational effort necessary to solve the even-parity problem with increasing numbers of arguments with our chosen population size of 4,000. When the even-5-parity function was run with a population size of 4,000 and each run arbitrarily stopped at our chosen maximum number $G = 51$ of generations to be run, no solution was found after 20 runs. Even after increasing the population size to 8,000 (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 dramatically increased with an increasing number of arguments (i.e. structural complexity was 45 and 149, respectively, above for the 3- and 4-parity functions).

The population size of 4,000 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.

3 AUTOMATIC FUNCTION DEFINITION

The inevitable increase in computational effort and structural complexity for solving parity problems of order greater than four 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 parameterized by dummy variables that would be helpful in decomposing and solving the problem.

If a human programmer were writing code for the even-3-parity or even-4-parity functions, he would probably choose to call upon either the odd-2-parity function (also known as the exclusive-or function `XOR`, the inequality function, and two-argument Boolean rule 6) or the even-2-parity function (also known as the equivalence function `EQV`, the not-exclusive-or function, and two-argument Boolean rule 9). If a human programmer were writing code for the even-5-parity function and parity functions with additional arguments, he would probably also want to call upon either the even-3-parity (three-argument Boolean rule 105) or the odd-3-parity (three-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 are, of course, in the 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]. When we talk about "function definition" in this paper, 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 *parameterized* by dummy variables (formal parameters). Specifically, if the exclusive-or function `XOR` were to be dynamically defined during a run, it would be a version of `XOR` parameterized by two dummy variables (called `ARG0` and `ARG1`), not a mere call to `XOR` with particular fixed terminals (e.g., `D0` and `D1`). When this parameterized 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 dynamically evolve 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 automatically and dynamically define functional subunits during runs (the specific terminology, of course, being specific to the particular paradigm). For example, when a set of weights are 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 zero or one). Note, however, that the function thus defined is called only once from 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.

3.1 EVEN-4-PARITY FUNCTION

Automatic function definition can be implemented within the context of genetic programming by establishing a constrained syntactic structure [Koza 1992, 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 there are only four rather impotent one-argument Boolean 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 3 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" (as

defined in Koza 1992, Chapter 19) 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, and (11) the body of the "main" result-producing branch.

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

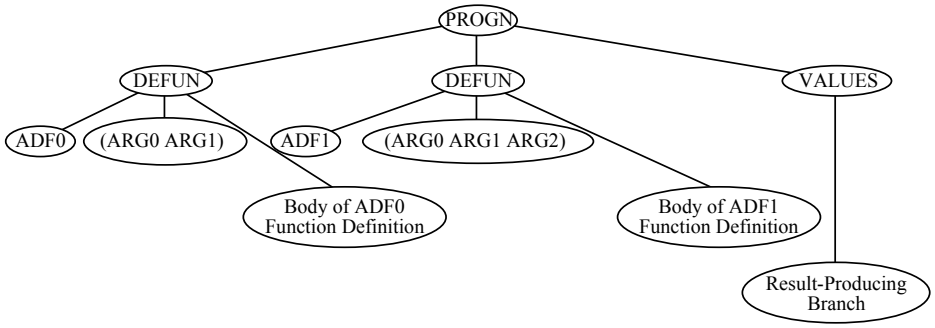

Figure 3 Abstraction of the overall structure of an S-expression with two function-defining branches and the one result-producing branch.

First, the body of ADF0 (i.e., the points of type 9) is a composition of functions from the given function set F and terminals from the terminal set A_2 of two dummy variables, namely A_2 = {ARG0, ARG1}.

Second, the body of ADF1 (i.e., the points of type 10) is a composition of functions from the given function set F and terminals from the set A_3 of three dummy variables, namely A_3 = {ARG0, ARG1, ARG2}.

Third, the body of the result-producing branch (i.e., the points of type 11) 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 F_3. F_3 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 F_3 is

$$F_3 = \{\text{AND, OR, NAND, NOR, ADF0, ADF1}\},$$

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 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.

Figure 4 shows an expansion of figure 3 representing a hypothetical program for the even-4-parity function containing two defined functions, ADF0 and ADF1. The first branch of the illustrative program in figure 4 contains a function definition for the two-argument defined function ADF0. This function is expressed in terms of the two dummy variables ARG0 and ARG1 and happens to be the even-2-parity function (i.e., the equivalence function EQV).

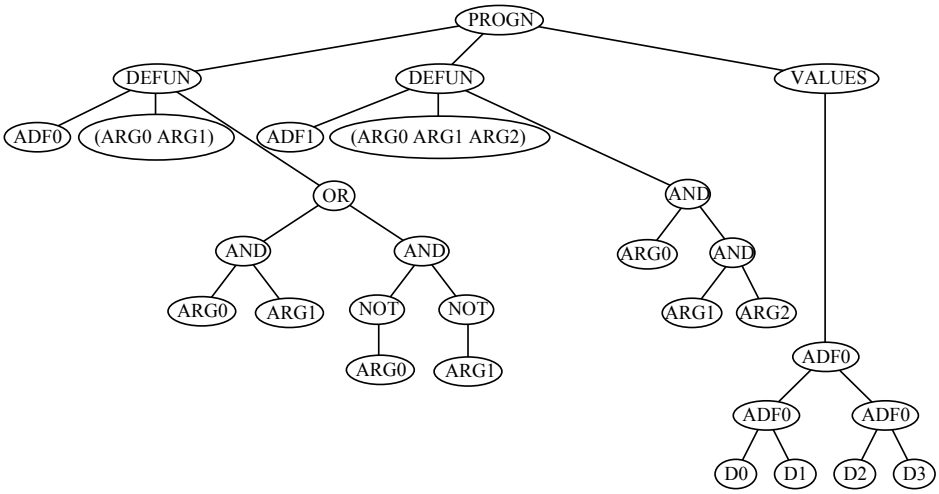

Figure 4 Illustrative program for the even-4-parity function containing two function definitions.

The second branch contains a function definition for the three-argument defined function ADF1. This function is expressed in terms of the three dummy variables ARG0, ARG1, and ARG2 and happens to be the ordinary three-argument conjunction function. The result-producing branch contains the actual variables of the problem (i.e., terminals) D0, D1, D2, and D3 and three calls to the defined function ADF0 (i.e., the equivalence function EQV) and no calls to the defined function ADF1. The result-producing branch computes

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

When the entire program is evaluated, the value returned is the value of the even-4-parity function.

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 (it being, of course, possible to solve this problem without any function definition by evolving the correct program in the result-producing branch). We do not favor 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.

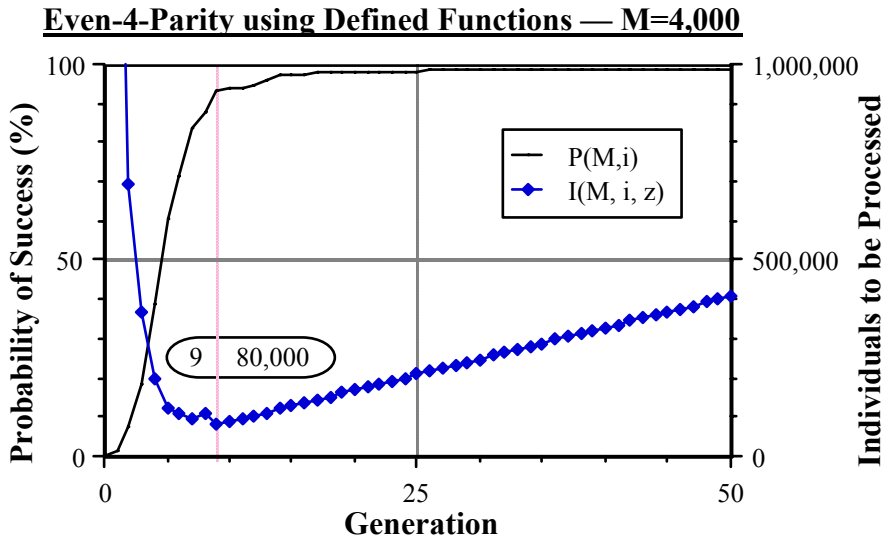

Figure 5 Performance curves for the even-4-parity problem showing that it is sufficient to process 80,000 individuals to yield a solution with automatic function definition.

In addition, 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 [Koza 1992, Chapter 19] 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. For example, if the crossover point of

the first parent comes from the body of ADF0 (type 9), a crossover fragment from the body of ADF1 (type 10) might contain the dummy variable ARG2, which is not an allowable point in the body of ADF0.

In one run of the even-4-parity problem, the best-of-generation individual from the initial random generation (i.e., generation 0) contained 23 points and had a raw fitness of 10 (out of a possible 16). As one would expect, generation 0 contains individuals with useless function definitions and equally useless result-producing branches. For example, ADF0 in the above best-of-generation individual was defined as the ordinary two-argument disjunction function (which is already in the original function set F).

Each new generation of the population 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 practical reasons of computer implementation, the depth of initial random programs was limited to 6 and the depth of programs created by crossover was limited to 17. The selection of values of the other parameters for this problem are the default values used on most of the other problems cited in Koza [1992].

In generation 12, the following S-expression appeared containing 74 points and attaining a perfect value of 16 for raw fitness:

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

The first branch of this best-of-run S-expression is a function definition for the two-argument defined function ADF0, which, when simplified, is equivalent to the exclusive-or (XOR) function (i.e. the odd-2-parity function).

The second branch defines the three-argument defined function ADF1, but this defined function is not called by the result-producing branch.

The result-producing branch of this best-of-run individual contains two references to ADF0. Upon substitution of XOR for ADF0, it becomes

```
(XOR (XOR D0 D2) (NAND (OR D3 D1) (NAND D1 D3))),
```

which, when simplified, is equivalent to

```
(XOR (XOR D0 D2) (EQV D3 D1)),
```

which is a correct solution to the even-4-parity problem.

Note that we did not specify that the exclusive-or function would be defined in ADF0, as opposed to, say, the equivalence function, the if-then function, or some other function. We did not specify that the exclusive-or function would be defined in the first branch as

opposed to the second branch. We did not specify that the second branch would be ignored. Genetic programming created the two-argument defined function ADF0 on its own in the first branch to help solve this problem. Having done this, genetic programming then used ADF0 in an appropriate way in the result-producing branch to solve the problem. Notice that the 41 points above are considerably fewer than the 149 points contained in the S-expression cited earlier for the even-4-parity problem.

Figure 5 presents the performance curves based on 168 runs for the even-4-parity function with automatic function definition. The cumulative probability of success $P(M,i)$ was 93% by generation 9 and was 99% by generation 50. The two numbers in the oval indicate that if this problem is run through to generation 9, processing a total of 80,000 individuals (i.e., 4,000 ∞ 10 generations ∞ 2 runs) is sufficient to yield a solution to this problem with 99% probability.

The 80,000 individuals that had to be processed for the even-4-parity problem using automatic function definition is one sixteenth of the 1,276,000 individuals needed when automatic function definition was not used (as shown in figure 2).

3.2 EVEN-5-PARITY FUNCTION

If automatic function definition is used, solutions to the even-5-parity and other higher order parity functions are readily found with a population size of 4,000.

For the even-5-parity problem, each S-expression has four branches with automatic function definition. The first three branches permit creation of function definitions with two, three, and four dummy variables. The result-producing (i.e., fourth) branch is an S-expression incorporating the four Boolean functions from the function set F; the three defined functions ADF0, ADF1, and ADF2; and the five terminals D0, D1, D2, D3, and D4.

In one run of the even-5-parity problem, genetic programming created a function definition for the even-3-parity function and created the necessary combination of operations in its result-producing branch to perform the behavior of the odd-2-parity function XOR.

Figure 6 graphically depicts the result-producing branch of this best-of-run individual.

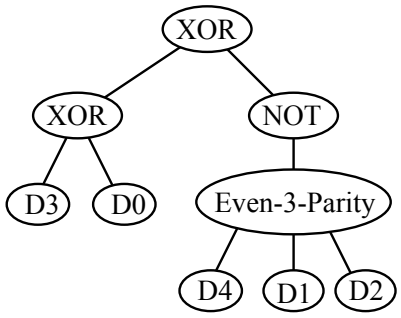

Figure 6 Result-producing branch of the best-of-run individual for the even-5-parity problem with automatic function definition.

Figure 7 shows the performance curves based on 7 runs for the even-5-parity function with automatic function definition. The cumulative probability of success $P(M,i)$ is 100% by generation 37. The two numbers in the oval indicate that if this problem is run through to generation 37, processing a total of 152,000 individuals (i.e., 4,000 ∞ 38 generations ∞ 1 run) is sufficient to yield a solution to this problem with 99% probability.

The 152,000 individuals that must be processed for the even-5-parity problem using automatic function definition is less than an eighth of the 1,276,000 individuals shown in figure 2 for the even-parity function of *only four* arguments.

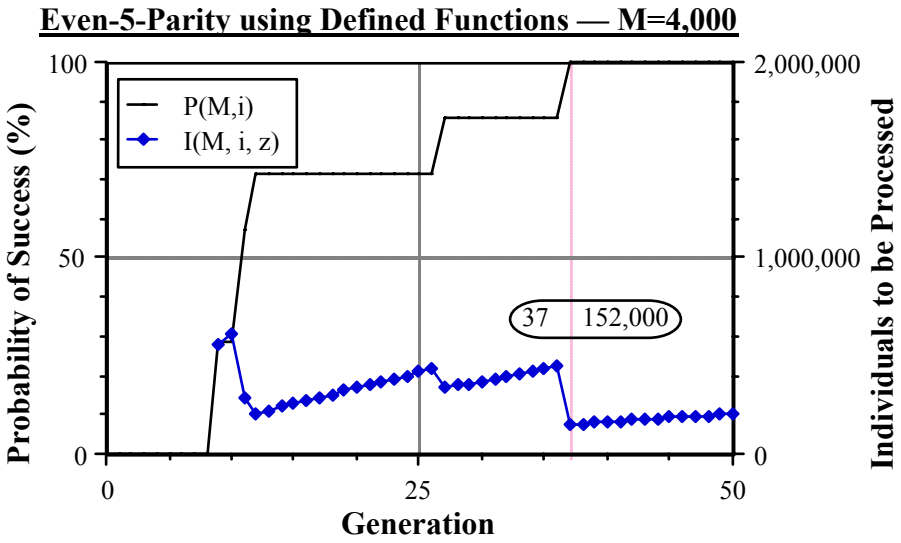

Figure 7 Performance curves for the even-5-parity problem show that it is sufficient to process 152,000 individuals to yield a solution with automatic function definition.

3.3 EVEN-6-PARITY FUNCTION

Similarly, the even-6-parity problem can be solved using automatic function definition. Processing a total of 812,000 individuals is sufficient to yield a solution to this problem with 99% probability. This 812,000 is less than the 1,276,000 individuals shown in figure 2 for the even-parity function of *only four* arguments.

In summary, automatic function definition considerably improved performance for the 4-, 5-, and 6-parity problems.

4 HIERARCHICAL AUTOMATIC FUNCTION DEFINITION

In the previous section on automatic function definition, the definition of a particular defined function never included a call to another defined function. However, it is common in ordinary programming to define one function in terms of other already-defined functions.

In the *hierarchical* form of automatic function definition, any function definition can call upon any other already-defined function. That is, there is a hierarchy (lattice) of function definitions wherein any function can be defined in terms of any combination of already-defined functions.

The number of function-defining branches and the number of dummy variables to appear in each function-defining branch is a matter of computer resources. We decided to use two function-defining branches and one result-producing branch. We also decided that, if the problem involves n terminals, then all function-defining branches will have $n - 1$ dummy variables; however, after the even-6-parity function, we reverted to four dummy

variables to reduce the bushiness of the S-expression and thereby save computer resources.

4.1 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 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 above S-expression calls upon the defined function ADF0 (i.e., XOR) to define ADF1. This second branch appears 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)).
```

Then, this result-producing branch 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 8 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).

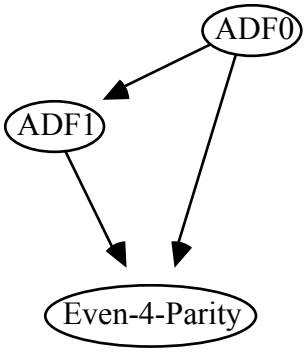

Figure 8 Hierarchy (lattice) of function definitions.

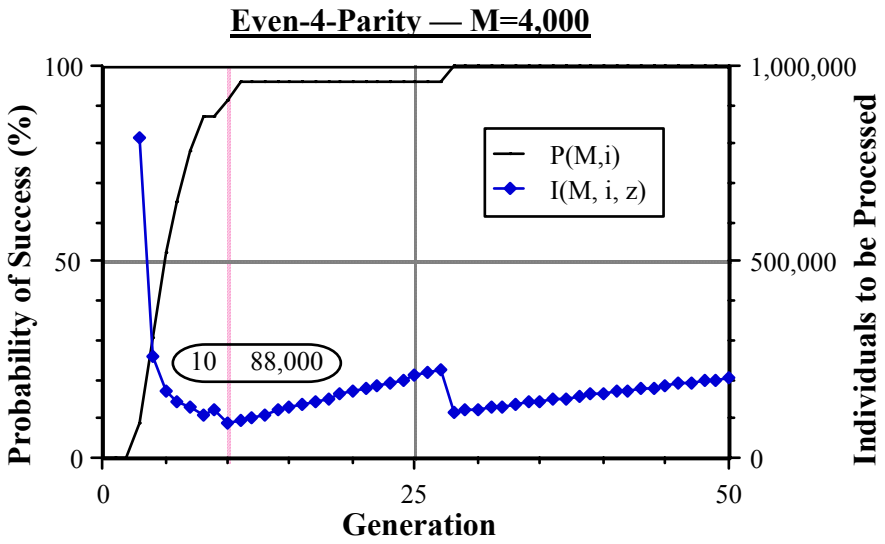

Figure 9 Performance curves for the even-4-parity problem show that it is sufficient to process 88,000 individuals to yield a solution with hierarchical automatic function definition.

Figure 9 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., 4,000 ∞ 11 generations ∞ 2 runs) is sufficient to yield a solution to this problem with 99% probability.

4.2 EVEN-5-PARITY FUNCTION

In one run of the even-5-parity problem, the following 100%-correct solution containing 160 points with a perfect value of raw fitness of 64 emerged on generation 12:

```
(PROGN (DEFUN ADF0 (ARG0 ARG1 ARG2 ARG3)
        (OR (OR (NOR (NOR ARG3 ARG1) (OR ARG1 ARG3)) (AND (NAND
        ARG1 ARG3) (NOR ARG1 ARG2))) (NAND (AND (OR ARG1 ARG2)
        (NAND ARG1 ARG2)) (NAND ARG1 (AND (NOR ARG3 ARG1)
        ARG0)))))
       (DEFUN ADF1 (ARG0 ARG1 ARG2 ARG3)
        (NAND (NAND (AND (NAND ARG1 ARG2) (ADF0 ARG0 ARG3 ARG0
        ARG2)) (NOR (NAND ARG3 ARG1) (AND ARG1 ARG1))) (AND (ADF0
        ARG0 (NAND ARG1 ARG2) (ADF0 ARG3 ARG0 ARG3 ARG0) (AND ARG1
        ARG1)) (ADF0 (ADF0 ARG3 ARG2 ARG3 ARG0) (ADF0 ARG0 ARG2
        ARG2 ARG1) (ADF0 ARG3 ARG3 ARG3 ARG0) (NOR ARG3 ARG0)))))
       (VALUES
        (OR (OR (NOR (ADF0 D3 D1 D1 D3) (OR D0 D1)) (NOR (NAND D1
        D2) (OR (OR D3 D2) (NOR D4 D4)))) (ADF1 (ADF1 D4 D0 D4 D1)
        (OR (OR (NOR (OR (NAND D1 D0) (ADF1 D1 D2 D3 D1)) (AND D4
        D0)) D2) (NOR (OR (NAND D1 D0) (ADF1 D1 D2 D3 D1)) (AND D4
        D0))) (NAND (ADF1 D1 D0 D0 D1) (NAND D0 D2)) (NAND (ADF1
        D3 D4 D0 D0) (ADF0 D3 D1 D1 D3))))))).
```

The first branch is equivalent to the four-argument Boolean rule 50,115, which is

```
(EQV ARG2 ARG1),
```

and 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

```
(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.

Figure 10 presents the performance curves based on 11 runs for the even-5-parity function with hierarchical automatic function definition. The cumulative probability of success $P(M,i)$ is 100% by generation 35. The two numbers in the oval indicate that if this problem is run through to generation 35, processing a total of 144,000 individuals (i.e., 4,000 ∞ 36 generations ∞ 1 run) is sufficient to yield a solution to this problem with 99% probability.

The number of individuals that must be processed to solve this problem with hierarchical automatic function definition (i.e., 144,000) is smaller than the 152,000 shown in figure 7, where only the non-hierarchical version of automatic function definition was employed.

4.3 EVEN 6-, 7-, 8-, 9-, AND 10-PARITY FUNCTIONS

The even 6- and 7-parity problems can be similarly solved using hierarchical automatic function definition. Processing a total of 864,000 individuals is sufficient to yield a solution to the even-6-parity problem with 99% probability. 1,440,000 individuals is sufficient to yield a solution to the even-7-parity problem with 99% probability.

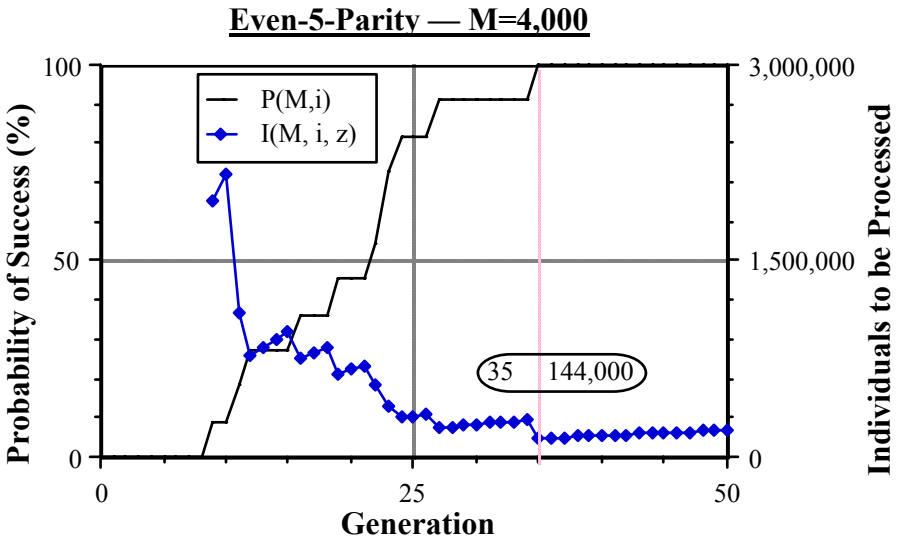

Figure 10 Performance curves for the even-5-parity problem using hierarchical automatic function definition show that it is sufficient to process 144,000 individuals.

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.

For example, in one run of the even-8-parity function, the best-of-generation individual containing 186 points and attaining a perfect value of raw fitness of 256 appeared in generation 24. The first branch of this S-expression defined a four-argument defined function `ADF0` (four-argument Boolean rule 10,280). The second branch of this S-expression defined a four-argument defined function `ADF1` (four-argument Boolean rule 26,214) which ignored two of its four arguments and is equivalent to

```
(XOR D0 D1).
```

In one run of the even-9-parity function, the best-of-generation individual containing 224 points and attaining a perfect value of raw fitness of 512 appeared in generation 40. The first branch of this S-expression defined a four-argument defined function `ADF0` (four-argument Boolean rule 1,872). The second branch of this S-expression defined a four-argument defined function `ADF1` (four-argument Boolean rule 27,030) which is equivalent to the odd-4-parity function.

In one run of the even-10-parity function, the best-of-generation individual containing 200 points and attaining a perfect value of raw fitness of 1,024 appeared in generation 40. The first branch of this S-expression defined a four-argument defined function `ADF0` (four-argument Boolean rule 38,791). The second branch of this S-expression defined a four-argument defined function `ADF1` (four-argument Boolean rule 23,205) which ignored one of its four arguments. This rule is equivalent to

```
(EVEN-3-PARITY D3 D2 D0).
```

4.4 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 2,048 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))))))).
```

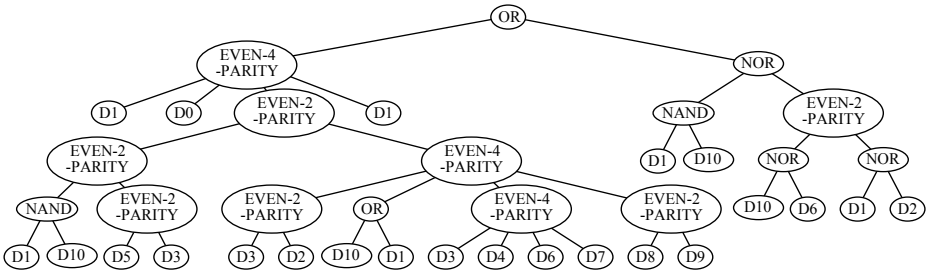

Figure 11 The best-of-run individual from generation 21 of one run of the even-11-parity problem is a composition of even-2-parity and even-4-parity functions.

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 the program shown below.

```
(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 contains only 220 points (compared to 347 points for the solution to the even-5-parity using genetic programming without any function definitions).

Figure 11 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.

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 $_{2}2{,}048 \sim _{10}616$.

A videotape visualization of the solution to the even-11-parity problem (and other problems involving genetic programming) can be found in Koza and Rice [1992].

In other words, the even-11-parity problem was solved by decomposing into parity functions of lower orders.

5 BOOLEAN 11-MULTIPLEXER

We can similarly learn other Boolean functions, such as the Boolean 11-multiplexer function (described in Koza 1991). For example, in one run with hierarchical automatic function definition, the program shown below appeared in generation 18.

```
(PROGN (DEFUN ADF0 (ARG0 ARG1 ARG2 ARG3)
         (NOR (OR ARG1 (NOR (NOR ARG2 ARG2)
                (NAND (NAND ARG3 ARG3) (OR ARG0 ARG2))))
           (AND ARG3 ARG0)))
       (DEFUN ADF1 (ARG0 ARG1 ARG2 ARG3)
         (NOR (AND ARG0 ARG3)
           (AND (OR ARG1 ARG2) (NAND ARG1 ARG3))))
       (VALUES
         (ADF0 (ADF1 D2 D0 A0 A1)
           (AND D3 A2)
           (NAND (OR (NOR (AND D3 A2) A1) D3) (OR D1 A1))
           (ADF1 D2 A0 (AND D3 A2) A2)))).
```

The first branch of this 56-point best-of-run individual defines the four-argument defined function `ADF0` which is equivalent to Boolean 4-argument rule 4,355. The second branch defines function `ADF1` which is equivalent to 4-argument Boolean rule 17,667. The third branch calls on `ADF0` once and calls `ADF1` twice in order to create the Boolean 11-multiplexer.

6 CONCLUSIONS

The first conclusion of this paper is that the problems of learning the even-parity functions and the Boolean multiplexer function can be solved with the newly developed technique of hierarchical automatic function definition in the context of genetic programming.

The second conclusion is that 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. As can be seen in table 1, automatic function definition and hierarchical automatic function definition are helpful in solving the even-parity problems.

Table 1 Number of individuals $I(M,i,z)$ required to be processed to yield a solution to various even-parity problems with 99% probability.

Even-parity function	Genetic Programming	Genetic Programming with automatic function definition	Genetic Programming with hierarchical automatic function definition
3	80,000		
4	1,276,000	80,000	88,000
5		152,000	144,000
6		812,000	864,000
7			1,440,000

6.1 Acknowledgments

James P. Rice of the Knowledge Systems Laboratory at Stanford University made numerous contributions in connection with the computer programming of the above.

6.1.1 References

Gruau, Frederic. Genetic synthesis of Boolean neural networks with a cell rewriting developmental process. In Schaffer, J. D. and Whitley, Darrell (editors).

Proceedings of the Workshop on Combinations of Genetic Algorithms and Neural Networks 1992. The IEEE Computer Society Press. 1992.

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

Koza, John R. A hierarchical approach to learning the Boolean multiplexer function. In Rawlins, Gregory (editor). *Foundations of Genetic Algorithms*. San Mateo, CA: Morgan Kaufmann Publishers Inc. 1991. Pages 171-192.

Koza, John R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection and Genetics*. Cambridge, MA: The MIT Press 1992.

Koza, John R. and Rice, James P. *Genetic Programming: The Movie.* Cambridge, MA: The MIT Press 1992.

In another run of the even-4-parity problem, both the first and the second branches were function definitions for the exclusive-or (XOR) function and the result-producing branch used both ADF0 and ADF1 in solving the problem. The three-argument defined function ADF1 in the second branch was degenerate in that it ignored one of the three available dummy variables. In another run, the ordinary AND function was defined in one of the function-defining branches and then used (along with the original primitive AND function from the function set F) in the result-producing branch. Although a human programmer would not usually write two equivalent function definitions for XOR or redefine an already-available primitive function, these genetically produced solutions are as good as the first solution above from the point of view of the fitness measure that drives the evolutionary process.

6.2 EVEN-6-PARITY FUNCTION

For the even-6-parity function, figure --- presents the performance curves based on 20 runs with automatic function definition. The cumulative probability of success *P(M,i)* is 50% by generation 28 and 55% by generation 50. The two numbers in the oval indicate that if this problem is run through to generation 28, processing a total of 812,000 (i.e., 4,000 ∞ 29 generations ∞ 7 runs) individuals is sufficient to yield a solution to this problem with 99% probability.

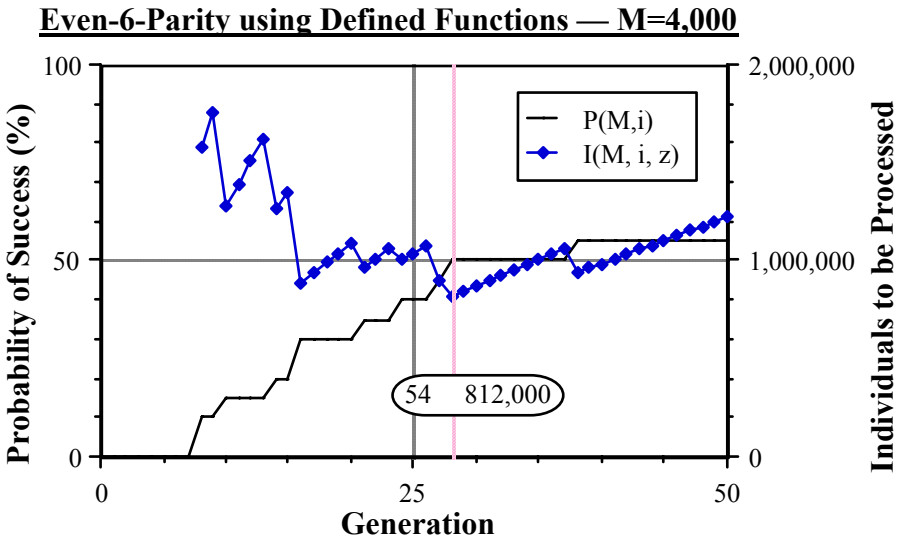

Figure --- Performance curves for the even-6-parity problem show that it is sufficient to process 812,000 individuals to yield a solution with automatic function definition.

The 812,000 individuals that must be processed for the even-6-parity problem using automatic function definition is less than the 1,276,000 individuals shown in figure 2 for the even-parity function of *only four* arguments.

In one run of the even-6-parity function (six-argument Boolean rule 10,838,310,072,981,296,745), the following best-of-generation individual containing 106 points and attaining a perfect value of raw fitness of 64 appeared in generation 7:

```
(PROGN (DEFUN ADF0 (ARG0 ARG1 ARG2 ARG3 ARG4)
         (OR (AND (AND ARG1 ARG2) (NAND ARG0 ARG4)) (NAND (NAND
         (AND ARG0 ARG1) ARG2) (OR ARG2 ARG1))))
       (DEFUN ADF0 (ARG0 ARG1 ARG2 ARG3 ARG4)
         (OR (ADF0 ARG4 ARG4 ARG3 ARG4 ARG3) (NOR ARG1 (OR ARG3
         ARG4))))
       (VALUES
         (ADF0 (AND D4 D0) (ADF0 (AND D4 D5) (NAND D0 D2) (ADF1 D3
         D5 D5 D3 D1) (OR D2 D0) (ADF0 D0 D0 D5 D4 D0)) (ADF1 (AND
         D1 D3) (AND D2 D0) (NAND D5 D5) (ADF0 D4 (NOR D5 D5) D4 D2
         D2) (OR D2 D0)) (OR (AND D4 D1) (NAND D3 D5)) (ADF0 (NOR
         D1 D3) (NAND D0 D5) (ADF0 D1 D5 D0 D0 D1) (ADF1 D3 D0 D3
         D2 D1) (AND D5 D1))))).
```

The first branch of this S-expression defined a five-argument defined function `ADF0` (five-argument Boolean rule 328,386,755) which ignored three of its five arguments and is equivalent to

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

The second branch defined a five-argument defined function ADF1 (five-argument Boolean rule 4,278,190,335) which also ignored three of its five arguments and is equivalent to

```
(XOR ARG3 ARG4).
```

Figure 11 presents the performance curves based on 37 runs for the even-6-parity function for hierarchical automatic function definition. The cumulative probability of success P(M,i) is 41% by generation 23, and 51% by generation 44. The two numbers in the oval indicate that if this problem is run through to generation 23, processing a total of 864,000 (i.e., 4,000 ∞ 24 generations ∞ 9 runs) individuals is sufficient to yield a solution to this problem with 99% probability.

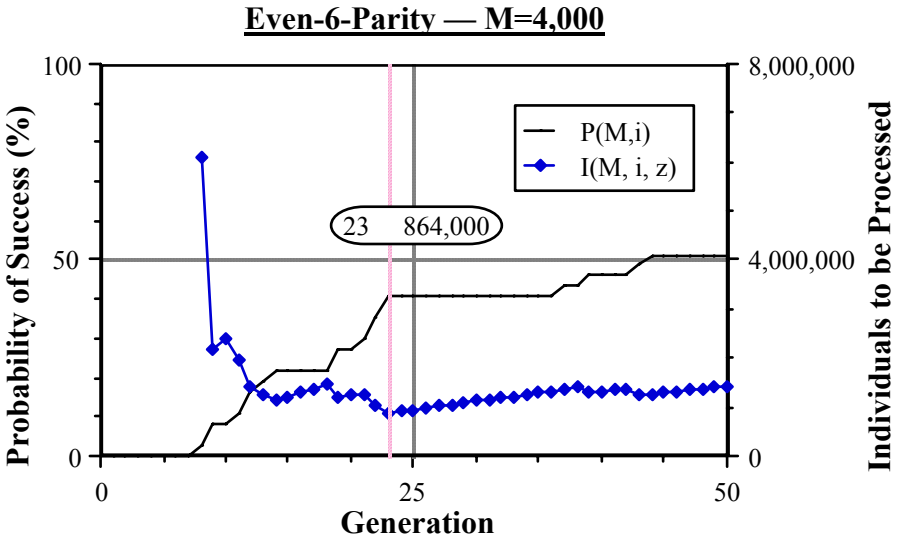

Figure 11 Performance curves for the even-6-parity problem using hierarchical automatic function definition show that it is sufficient to process 864,000 individuals to yield a solution.

In one run of the even-7-parity function, the best-of-generation individual containing 92 points and attaining a perfect value of raw fitness of 128 appeared in generation 14. The first branch of this S-expression defined a four-argument defined function ADF0 (four-argument Boolean rule 42,245). The second branch of this S-expression defined a four-argument defined function ADF1 (four-argument Boolean rule 49,980) which ignored one of its four arguments. ADF1 is equivalent to

```
(ODD-3-PARITY D3 D2 D1).
```

Figure 12 presents the performance curves based on 29 runs for the even-7-parity function for hierarchical automatic function definition. The cumulative probability of

success P(M,i) is 20.7% by generation 17, and 34.5% by generation 50. The two numbers in the oval indicate that if this problem is run through to generation 17, processing a total of 1,440,000 (i.e., 4,000 ∞ 18 generations ∞ 20 runs) individuals is sufficient to yield a solution to this problem with 99% probability.

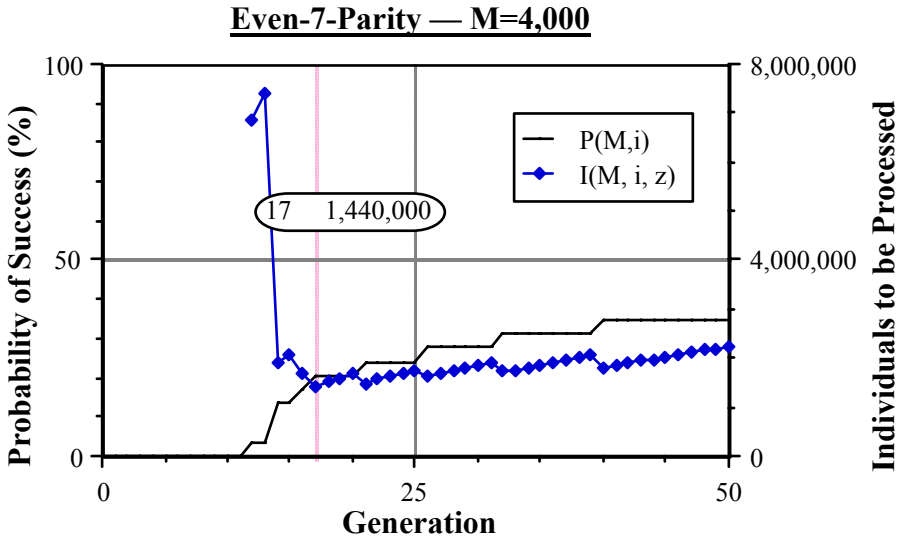

Figure 12 Performance curves for the even-7-parity problem using hierarchical automatic function definition show that it is sufficient to process 1,440,000 individuals to yield a solution.
