DISCOVERY OF A MAIN PROGRAM AND REUSABLE SUBROUTINES USING GENETIC PROGRAMMING

John R. Koza
Computer Science Department
Margaret Jacks Hall
Stanford University
Stanford, California 94305-2140
Koza@CS.Stanford.Edu
415-941-0336

ABSTRACT

This paper describes an approach for automatically decomposing a problem into subproblems, automatically creating reusable subroutines to solve the subproblems, and automatically assembling the results produced by the subroutines in order to solve the problem. The approach uses genetic programming with the recently developed additional facility of automatic function definition. Genetic programming provides a way to genetically breed a computer program to solve a problem and automatic function definition enables genetic programming to create reusable subroutines dynamically during a run. The approach is applied to an illustrative problem containing a considerable amount of regularity. Solutions to the problem produced using automatic function definition are considerably smaller in size and require processing of considerably fewer individuals than is the case without automatic function definition. Specifically, the average program size for a solution to the problem without using automatic function definition is 3.65 times larger than the size for a solution when using automatic function definition. The number of individuals required to be processed to yield a solution with 99% probability without automatic function definition is 9.09 times larger than the equivalent number required with automatic function definition.

1. INTRODUCTION AND OVERVIEW

A three-step hierarchical approach is often advocated for automating the solution to complex problems. In this process, one first tries to discover a way to decompose a given problem into subproblems. Second, one tries to solve each of the presumably simpler subproblems. Third, one seeks a way to assemble the solutions to the subproblems into a solution to the original overall problem.

To the extent that the problem environment contains regularities, it may be possible to decompose a given overall problem in such a way that the solutions to some of the subproblems are reusable many times in assembling the solution to the overall problem. Repeated use of subproblem solutions promises the potential of producing a solution to the overall problem that is smaller in size than an approach that fails to exploit such regularities of the problem environment. Such reuse should also produce a solution to the overall problem requiring the processing of fewer individuals as part of the automated problem-solving process. Thus, reuse of subproblem solutions should potentially provide leverage to enable machine learning methods to be scaled up from small "proof of principle" problems to evermore complex tasks.

This paper describes a way for simultaneously discovering reusable subroutines and a way of assembling calls to the reusable subroutines in order to solve a problem. Specifically, this paper describe a problem-solving process that

- automatically decomposes a problem into subproblems,
- automatically discovers the solution to the subproblems, and
- automatically discovers a way to assemble the solutions of the subproblems into a solution of the overall problem.

The approach involves using genetic programming with the recently developed additional facility of automatic function definition.

Genetic programming provides a way to search the space of computer programs composed of certain terminals and primitive functions to find a function which solves, or approximately solves, a problem. Automatic function definition enables genetic programming to define potentially reusable functions automatically and dynamically during a run and also to assemble calls to these defined functions dynamically during a run in order to solve a problem.

Section 2 of this paper describes genetic programming and section 3 describes automatic function definition. Section 4 presents an illustrative problem. Section 5 details the preparatory steps for applying genetic programming without automatic function definition to the problem. The problem is solved in section 6 without automatic function definition. Section 7 details the preparatory steps with

automatic function definition and section 8 solves the problem with automatic function definition. The two approaches are compared in the concluding section 9.

2. BACKGROUND

Since John Holland first presented the genetic algorithm in detail in his pioneering *Adaptation in Natural and Artificial Systems* (1975), the genetic algorithm has proven successful at finding an optimal point in a search space for a wide variety of problems (Goldberg 1989, Davis 1987, Davis 1991, Michalewicz 1992).

Genetic programming is an extension of the genetic algorithm in which the genetic population consists of computer programs (that is, compositions of primitive functions and terminals).

Genetic programming provides a way to search the space of programs to find a function which solves, or approximately solves, a problem. The book *Genetic Programming: On the Programming of Computers by Means of Natural Selection* (Koza 1992a) describes genetic programming and demonstrates that populations of computer programs can be genetically bred to solve a surprising variety of problems in a wide variety of fields. A videotape visualization of numerous applications of genetic programming can be found in the *Genetic Programming: The Movie* (Koza and Rice 1992).

Genetic programming is a domain independent method that genetically breeds populations of 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 sub-steps until the termination criterion has been satisfied:
    (a) Execute each program in the population and assign it a fitness value according to how well it solves the problem.
    (b) Create a new population of programs by applying the following two primary operations. The operations are applied to program(s) in the population selected with a probability based on fitness (i.e., the fitter the program, the more likely it is to be selected).
        (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. The genetic crossover (sexual recombination) operation (described below) operates on two parental computer programs and produces two offspring programs using parts of each parent.
(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.

Crossover Operation

The genetic crossover (sexual recombination) operation operates on two parental computer programs selected with a probability based on fitness and produces two new offspring programs consisting of parts of each parent.

For example, consider the following computer program (LISP symbolic expression):

`(+ (* 0.234 Z) (- X 0.789)),`

which we would ordinarily write as

$$0.234\ Z + X - 0.789.$$

This program takes two inputs (`X` and `Z`) and produces a single floating point output. In the prefix notation used in Lisp, the multiplication function `*` is first applied to the terminal `0.234` and the value of the terminal `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:

`(* (* Z Y) (+ Y (* 0.314 Z))),`

which is equivalent to

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

In figure 1, these two programs are depicted as rooted, point-labeled 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.

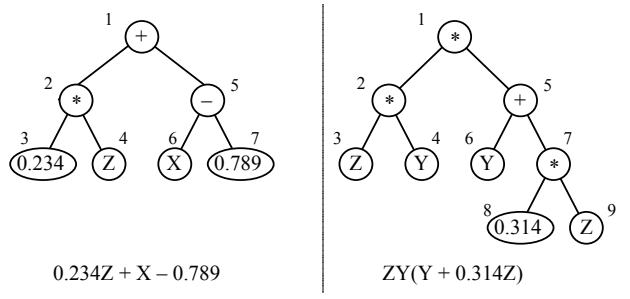

Figure 1 Two Parental computer programs.

The crossover operation creates new offspring by exchanging sub-trees (i.e., sub-lists, 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 the + in the second parent. The two crossover fragments are the two sub-trees shown in Figure 2.

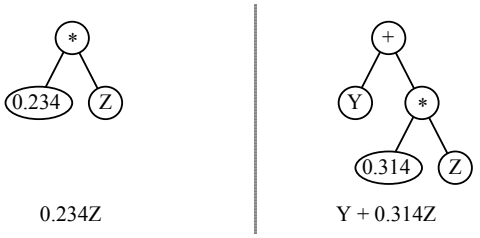

Figure 2 Two Crossover Fragments

These two crossover fragments correspond to the underlined sub-programs (sub-lists) in the two parental computer programs.

The two offspring resulting from crossover are

```
(+ (+ Y (* 0.314 Z)) (- X 0.789))
```

and

```
(* (* Z Y) (* 0.234 Z)).
```

The two offspring are shown in Figure 3.

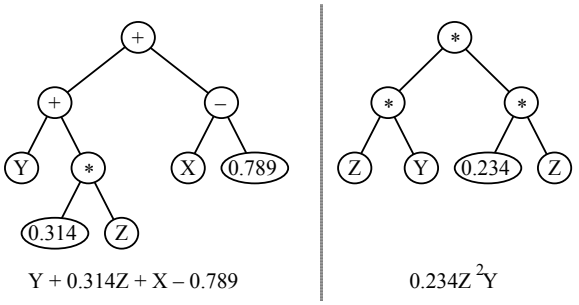

Figure 3 Two Offspring.

Thus, crossover creates new computer programs using parts of existing parental programs. Assuming closure among the given set of functions and terminals, this crossover operation produces syntactically and semantically valid programs as offspring. Because programs are selected to participate in the crossover operation with a probability based on fitness, crossover allocates future trials to regions of the search space whose programs contain parts from promising programs.

3. AUTOMATIC FUNCTION DEFINITION

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 writing a program for Boolean parity functions of several different orders might find it convenient first to write a subroutine for some lower-order parity function. The programmer would call on the code for this low-order parity function at different places and in different ways in his main program and combine the results 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 from the main program. Second, this function definition 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, this function definition contains a body which performs the work of the function. Fourth, this function definition identifies the value to be returned by the function. In this example, the single value to be returned is emphasized and highlighted using an explicit invocation of the `values` function.

This particular function definition has two dummy variables, 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 dummy variables from more than one place in the main program. For example, a programmer needing the even-4-parity function might write

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

The process of defining and calling a function, in effect, decomposes the problem into a hierarchy of subproblems. When this decomposition is performed so as to create reusable defined functions, the function definitions can

exploit the underlying regularities and symmetries of a problem by obviating the need to tediously rewrite lines of essentially similar code.

Automatic function definition can be implemented within the context of genetic programming by establishing a constrained syntactic structure for the individual programs in the population (Koza 1992a; Koza and Rice 1992). Each program in the population contains one (or more) function-defining branches and one (or more) "main" result-producing branches. A result-producing branch usually calls the defined functions from one or more places. A defined function may hierarchically refer to another already-defined function. A defined function may even refer to itself, although such recursive references are not discussed in this paper.

Figure 4 shows the overall structure of a program consisting of one function-defining branch and one result-producing branch. The function-defining branch appears in the left part of this figure and the result-producing branch appears on the right.

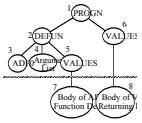

Figure 4 Program with one function-defining branch and one result-producing branch

There are eight different "types" of points in this program. The first six types are invariant and appear above the horizontal dotted line in this figure. The eight types are as follows:

(1) the root of the tree (which consists of the place-holding PROGN connective function),
(2) the top point, DEFUN, of the function-defining branch,
(3) the name, ADF0, of the automatically defined function,
(4) the argument list of the automatically defined function,
(5) the VALUES function of the function-defining branch identifying, for emphasis, the value(s) to be returned by the automatically defined function,
(6) the VALUES function of the result-producing branch identifying, for emphasis, the value(s) to be returned by the result-producing branch,
(7) the body (i.e., work) of the automatically defined function ADF0, and
(8) the body of the result-producing branch.

When the overall program is evaluated, the PROGN causes the sequential evaluation of the two branches. The function-defining branch merely defines the automatically defined function ADF0 and does not immediately return any useful value. The value(s) returned by the overall program consists only of the value(s) returned by the VALUES function associated with the result-producing branch. If there were a second function definition (defining ADF1), there would an additional function-defining branch containing points of types 2, 3, 4, and 5 and there would be an additional type 7a for points in the body of automatically defined function ADF1.

4. THE PROBLEM

After determining that genetic programming with automatic function definition could perform Boolean function learning on parity problems of various orders (Koza 1992a, 1992b), could discover an impulse response function of a time-invariant linear system (Koza, Keane, and Rice 1993), could create a pattern-recognizing program (Koza 1993a), and could generate a computer program for controlling the movement of an artificial ant so that the ant can find all the food lying along an irregular trail (Koza 1993b), the question arose as to whether this new technique was applicable to other types of problems where the problem environment contains exploitable regularities.

This paper explores this question in the context of a problem where the goal is to find a program for controlling the lawn mower so that the lawn mower cuts all the grass in the yard. The lawn mower operates in a 8 by 8 square area of lawn initially containing grass in all 64 squares. Each square is uniquely identified by a vector of integers modulo 8 of the form (i,j), where $0 \leq i, j \leq 7$. The lawn is toroidal in both directions, so that whenever the lawn mower moves off the side of the lawn, it reappears on the opposite side. The state of the lawn mower consists of its location on the lawn and the direction in which it is facing. The lawn mower starts at location (3,3) facing north.

The lawn mower is capable of turning left, of moving forward one square in the direction in which it is currently facing, and of jumping by a specified displacement in the vertical and horizontal directions. Whenever the lawn mower moves onto a new square (either by means of a single move or a jump), it mows the grass, if any, at the location onto which it arrives.

A human programmer writing a program to solve this problem would almost certainly not solve it by tediously writing a sequence of 64 separate mowing operations. Instead, a human programmer would exploit the considerable symmetry and regularity inherent in this problem environment by writing a program that mows a certain small area of the lawn in a particular way, repositioning the lawn mower in some efficient and regular way, and then repeating the particular mowing action on the new area of the lawn. That is, the human programmer would decompose the overall problem into a set of subproblems (i.e., mowing the small area), solve the subproblem, and then repeatedly reuse the subproblem solution in order to solve the overall problem.

5. PREPARATORY STEPS WITHOUT AUTOMATIC FUNCTION DEFINITION

There are five major steps in preparing to use genetic programming on any problem, namely determining

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

The terminal set for this problem consists of two side-effecting operators and random vector constants. That is,

$\mathcal{T}$ = {(LEFT), (MOW), ←}.

Each random constant ← consists of a vector #(i j) of integers modulo 8.

(LEFT) and (MOW) each are operators that take no explicit arguments, but have side effects of the state of the lawn mower. The operator (LEFT) rotates the orientation of the lawn mower left by 90° (without moving the lawn mower). (LEFT) returns the vector value (0,0).

The operator (MOW) moves the lawn mower in the direction it is currently facing and mows the grass, if any, in the square onto which it is moving (thereby removing grass, if any, from that square). (MOW) returns the vector value #(0 0). For example, if the lawn mower is at location (1,3) facing east, (MOW) increments the first component (i.e., the X location) of the state vector of the lawn mower thus moving the lawn mower to location (2,3) with the lawn mower still facing east. As a further example, if the lawn mower is at location (7,3) facing east, (MOW) moves the lawn mower to location (0,3) because of the toroidal geometry.

The function set consists of

$\mathcal{F}$ = {V+, FROG, PROGN},

with these functions taking 2, 1, and 2 arguments, respectively.

V+ is a two-argument vector addition modulo 8. For example, (V+ #(1 2) #(3 7)) returns the vector value #(4 1).

FROG is a one-argument operator that causes the lawn mower to move relative to the direction it is currently facing by an amount specified by its vector argument. For example, if the lawn mower is at location (1,2) and is facing east, (FROG #(3 5)) causes the lawn mower to end up at location (6,5) with the lawn mower still facing east. FROG acts as the identity operator on its argument, so in this example it would return #(3 5).

PROGN is a two-argument connective form that causes the execution of its two arguments in sequence and returns the value of the last argument.

Each branch of the overall program without automatic function definition is a composition of primitive functions from the function set $\mathcal{F}$ and terminals from the terminal set $\mathcal{T}$.

The third major step in preparing to use genetic programming is the identification of the fitness measure for evaluating the goodness of each individual in the population.

The lawn mower's goal is to mow all the grass in the 64 squares. The activity of the lawn mower is terminated when the lawn mower has executed either a total of 100 LEFT turns or 100 moving operations (i.e., a MOW or FROG). The raw fitness of a particular program is the amount of grass (from 0 to 64) mowed within the allowed amount of time.

The fourth major step in preparing to use genetic programming is the selection of values for certain parameters. The primary parameters in genetic programming are the population size and the maximum number of generations to be run. Our choice of 1,000 as the population size and our choice of 51 as the maximum number of generations to be run reflect an estimate on our part as to the likely difficulty of this problem. Our choice of values for the various secondary parameters that control a run of genetic programming are the same default values as we have consistently used on numerous other problems (Koza 1992a), except that we continue our recently adopted practice of using tournament selection (with a group size of seven) as the selection method (as opposed to fitness proportionate selection) and of not using greedy over-selection.

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. We will terminate a given run if we encounter an individual that mows all 64 areas of grass within the allotted time or after 51 generations. We designate the best individual obtained during the run (the best-so-far individual) as the result of the run.

6. RESULTS WITHOUT AUTOMATIC FUNCTION DEFINITION

In one particular and typical successful run of genetic programming without automatic function definition on this problem, the following 296-point individual scoring 64 hits emerged on generation 34:

```
(V+ (V+ (V+ (FROG (PROGN (PROGN (V+ (MOW) (MOW))
(FROG #(3 2))) (PROGN (V+ (PROGN (V+ (PROGN (PROGN
```

```
(MOW) #(2 4)) (FROG #(5 6))) (PROGN (V+ (MOW) #(6
0)) (FROG #(2 2)))) (V+ (MOW) (MOW))) (PROGN (V+
(PROGN (PROGN #(0 3) #(7 2)) (FROG #(5 6))) (PROGN
(V+ (MOW) #(6 0)) (FROG #(2 2)))) (V+ (MOW)
(MOW)))) (PROGN (FROG (MOW)) (PROGN (PROGN (PROGN
(V+ (MOW) (MOW)) (FROG (LEFT))) (PROGN (MOW) (V+
(MOW) (MOW)))) (PROGN (V+ (PROGN #(0 3) #(7 2))
(V+ (MOW) (MOW))) (PROGN (V+ (MOW) (MOW)) (PROGN
(LEFT) (MOW)))))))) (V+ (PROGN (V+ (PROGN (PROGN
(MOW) #(2 4)) (FROG #(5 6))) (PROGN (V+ (MOW) #(6
0)) (FROG #(2 2)))) (V+ (MOW) (MOW))) (V+ (FROG
(LEFT)) (FROG (MOW))))) (V+ (FROG (V+ (PROGN (V+
(PROGN (V+ (MOW) (MOW)) (FROG #(3 7))) (V+ (PROGN
(MOW) (LEFT)) (V+ (MOW) #(5 3)))) (PROGN (PROGN
(V+ (PROGN (LEFT) (MOW)) (V+ #(1 4) (LEFT)))
(PROGN (FROG (MOW)) (V+ (MOW) #(3 7)))) (V+ (PROGN
(FROG (MOW)) (V+ (LEFT) (MOW))) (V+ (FROG #(1 2))
(V+ (MOW) (LEFT)))))) (PROGN (V+ (FROG #(3 1)) (V+
(FROG (PROGN (PROGN (V+ (MOW) (MOW)) (FROG #(3
2))) (FROG (FROG #(5 0))))) (V+ (PROGN (FROG
(MOW)) (V+ (MOW) (MOW))) (V+ (FROG (LEFT)) (FROG
(MOW)))))) (PROGN (PROGN (PROGN (PROGN (LEFT)
(MOW)) (V+ (MOW) #(3 7))) (V+ (V+ (MOW) (MOW))
(PROGN (LEFT) (LEFT)))) (V+ (FROG (PROGN #(3 0)
(LEFT))) (V+ (PROGN (MOW) (LEFT)) (FROG #(5
4)))))))) (PROGN (FROG (V+ (PROGN (V+ (PROGN
(PROGN (V+ (PROGN (PROGN (MOW) #(2 4)) (FROG #(5
6))) (PROGN (V+ (MOW) #(1 2)) (FROG #(2 2)))) (V+
(MOW) (MOW))) (FROG #(3 7))) (V+ (PROGN (PROGN
(MOW) #(2 4)) (FROG #(5 6))) (PROGN (V+ (MOW) #(6
0)) (FROG #(2 2))))) (PROGN (PROGN (V+ (FROG
(MOW)) (V+ #(1 4) (LEFT))) (PROGN (FROG (MOW)) (V+
(MOW) #(3 7)))) (V+ (PROGN (FROG (MOW)) (V+ (LEFT)
(MOW))) (V+ (FROG #(1 2)) (V+ (MOW) (LEFT))))))
(PROGN (V+ (PROGN (FROG #(2 4)) (V+ (MOW) (MOW)))
(V+ (FROG (MOW)) (LEFT))) (PROGN #(3 0) (LEFT)))))
(FROG (V+ #(7 4) (MOW)))))) (V+ (V+ (PROGN (MOW)
#(4 3)) (V+ (LEFT) #(6 1))) (MOW)))
```

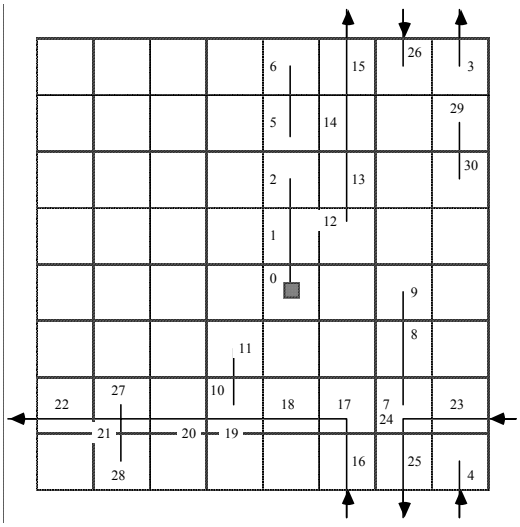

Figure 5 Partial trajectory for operations 0 through 30 of 296-point program without automatic function definition.

Figure 5 shows a partial trajectory of this best-of run 296-point individual for operations 0 through 30; figure 6 shows a partial trajectory for operations 31 through 60; and figure 7 shows a partial trajectory for operations 61 through 85. The overall trajectory is divided over three figures as a visual aid. As can be seen, even though the problem environment contains considerable regularity in that it requires mowing all 64 squares of the lawn, this solution operates in an entirely *ad hoc* fashion. For example, between operations 2 and 3, the lawn mower flits up two rows and three columns to the right, then goes up six and three to the left between operations 4 and 5, and then goes up two (i.e., down six) and two to the right between operations 6 and 7. This 296-point program solves the problem by agglomerating enough erratic movements so as to cover the entire area of the lawn.

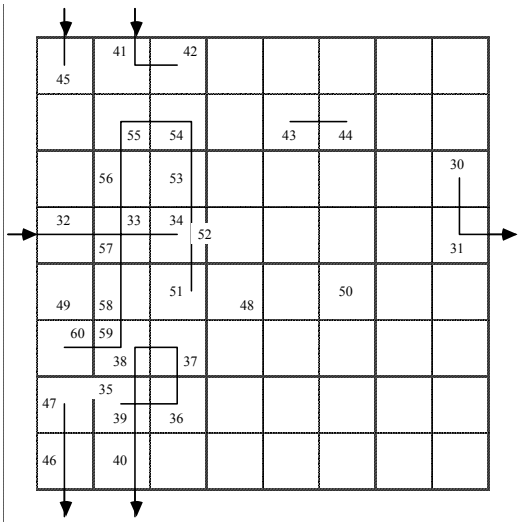

Figure 6 Partial trajectory for operations 31 through 60 of 296-point program without automatic function definition.

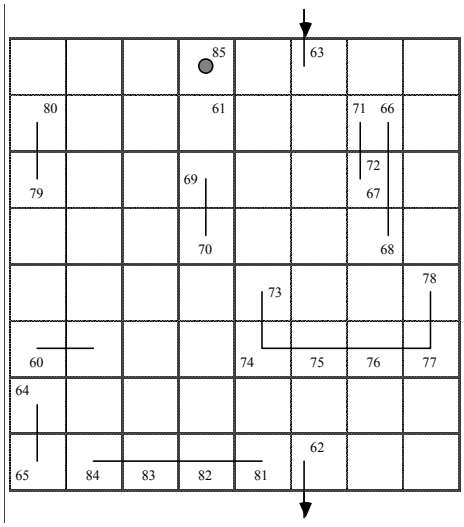

Figure 7 Partial trajectory for operations 61 through 85 of 296-point program without automatic function definition.

Over a series of 38 runs, the average structural complexity of the 35 successful solutions to the lawn mower problem without automatic function definition was 280.82 points.

The rising curve in figure 8 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 program in the population which scores 64). As can be seen, the experimentally observed value of $P(M,i)$ is 92% by generation 49, and 92% by generation 50 over the 38 runs.

The second curve in the figure (which starts falling from the upper left) shows, by generation, the number of individuals that must be processed, $I(M,i,z)$, to yield, with probability z = 99%, 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 required is given by

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

where the brackets indicate the ceiling function for rounding up to the next highest integer.

The $I(M,i,z)$ curve reaches a minimum value in the figure at generation 49 (highlighted by the light dotted vertical line). For a value of $P(M,i)$ of 92%, 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 (49 and 100,000) in the oval indicate that if this problem is run through to generation 49 (the initial random generation being counted as generation 0), processing a total of 100,000 individuals (i.e., 1,000 ∞ 50 generations ∞ 2 runs) is sufficient to yield a solution to this problem with 99% probability. This number, 100,000, is a measure of the computational effort necessary to yield a solution to this problem with 99% probability without automatic function definition.

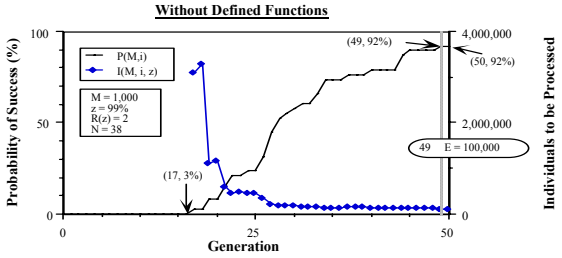

Figure 8 Performance curves showing that it is sufficient to process 100,000 individuals to yield a solution with 99% probability without automatic function definition.

7. PREPARATORY STEPS WITH AUTOMATIC FUNCTION DEFINITION

In applying genetic programming with automatic function definition to the lawn mower problem, we first decided that each individual overall program in the population will consist of two function-defining branches (defining a zero-argument function called `ADF0` and a one-argument function `ADF1`) and a final (rightmost) result-producing branch. Since `ADF0` is defined before `ADF1`, `ADF0` is allowed to hierarchically call `ADF0`.

We first consider `ADF0`, the first of the two automatically defined functions.

The terminal set $\mathcal{T}_{fd0}$ for the zero-argument defined function `ADF0` consists of

$$\mathcal{T}_{fd0} = \{\texttt{(LEFT)}, \texttt{(MOW)}, \leftarrow\}.$$

The function set $\mathcal{F}_{fd0}$ for the zero-argument defined function `ADF0` is

$$\mathcal{F}_{fd0} = \{\texttt{V+}, \texttt{PROGN}\},$$

each taking 2 arguments.

The body of `ADF0` is a composition of primitive functions from the function set $\mathcal{F}_{fd0}$ and terminals from the terminal set $\mathcal{T}_{fd0}$.

We now consider `ADF1`.

The terminal set $\mathcal{T}_{fd1}$ for the one-argument defined function `ADF1` taking dummy variable `ARG0` consists of

$$T_{fd1} = \{\texttt{ARG0}, \texttt{(LEFT)}, \texttt{(MOW)}, \leftarrow\}.$$

The function set $\mathcal{F}_{fd1}$ for the one-argument defined function `ADF1` is

$$\mathcal{F}_{fd1} = \{\texttt{ADF0}, \texttt{V+}, \texttt{FROG}, \texttt{PROGN}\},$$

taking 0, 2, 1, and 2 arguments, respectively,

The body of `ADF1` is a composition of primitive functions from the function set $\mathcal{F}_{fd1}$ and terminals from the terminal set $\mathcal{T}_{fd1}$.

Since `(LEFT)` and `(MOW)` each evaluate to `#(0 0)` and since `FROG` acts as an identity function returning its own argument, the value returned by `ADF0` and `ADF1` is either `#(0 0)` or the result of vector addition `V+` operating on random constants or random constants or `ARG0` in the case of `ADF1`.

We now consider the result-producing branch.

The terminal set $\mathcal{T}_{rp}$ for the result-producing branch is

$$\mathcal{T}_{rp} = \{\texttt{(LEFT)}, \texttt{(MOW)}, \leftarrow\}.$$

The function set $\mathcal{F}_{rp}$ for the result-producing branch is

$$\mathcal{F}_{rp} = \{\texttt{ADF0},\ \texttt{ADF1},\ \texttt{V+}, \texttt{FROG}, \texttt{PROGN}\},$$

with the functions taking 0, 1, 2, 1, and 2 arguments, respectively.

The result-producing branch is a composition of the functions from the function set $\mathcal{F}_{rp}$ and terminals from the terminal set $\mathcal{T}_{rp}$.

Since each individual program in the population consists of two function-defining branches and one result-producing branch, we create the initial random generation so that every individual program in the population has this particular constrained syntactic structure.

Since a constrained syntactic structure is involved, we must perform crossover so as to preserve the syntactic validity of all offspring as the run proceeds from generation to generation. To implement structure-preserving crossover, crossover is limited to points lying within the bodies of ADF0, ADF1, or the result-producing branch. The crossover point for the first parent is selected at random from one of these three bodies. However, once this selection is made, the crossover point of the second parent is selected at random from the same body from which the crossover point of the first parent was selected, i.e., the crossover points always share the same "type" as defined in Section 3.

As the run progresses, genetic programming will evolve different function definitions in the function-defining branches of each overall program and then, at its discretion, may call such defined functions from its result-producing branch. The structures of both the function-defining and the result-producing branch are determined by the combined effect, over many generations, of the selective pressure exerted by the fitness measure and by the effects of the operations of Darwinian fitness proportionate reproduction and crossover.

8. RESULTS WITH AUTOMATIC FUNCTION DEFINITION

In one particular successful run with automatic function definition, the following 100% correct 42-point program scoring 64 (out of 64) emerged in generation 5:

```
(progn  (defun ADF0 ()
          (values (PROGN (V+ #(0 1) #(2 0)) (V+ (V+
            (PROGN (MOW) (LEFT)) (V+ (MOW) (LEFT)))
            (PROGN (V+ (LEFT) (LEFT)) (PROGN (MOW)
            (MOW)))))))
        (defun ADF1 (ARG0)
          (values (V+ (FROG (FROG (ADF0))) (PROGN
            (PROGN (V+ (MOW) (ADF0)) (V+ (ADF0)
            (MOW))) (V+ (FROG (ADF0)) (V+ ARG0
            ARG0))))))
        (values (ADF1 (ADF1 (ADF1 (ADF1
         (ADF0)))))))
```

Note that this 42-point solution is a hierarchical decomposition of the problem. Genetic programming discovered the decomposition of the overall problem, discovered the content of each subroutine, and assembled the results of the multiple calls to the subroutines into a solution of the overall problem. Specifically, in the result-producing branch at the top level, genetic programming discovered a decomposition of the overall problem into five subproblems (four ADF1s and one ADF0). As it happens, the result-producing branch does not contain any (LEFT), (MOW), or FROG operations. ADF1 contains four invocations of ADF0, two (MOW)'s, and no (LEFT) or FROG operations. ADF0 contains four (MOW)'s, and four (LEFT)'s.

Figure 9 shows the trajectory of the lawn mower for this 42-point solution. Note the difference between this regular, largely non overlapping trajectory and the haphazard character of the three partial trajectories shown in figures 5, 6, and 7. The lawn mower here takes advantage of the regularity of the problem environment. It employs a tessellating activity that covers the entire lawn in a regular manner. Specifically, it mows four consecutive squares in a column in a northerly direction, shifts one column to the west, and then does the same thing in the next column. The fact that the entire trajectory can be conveniently presented in only one figure testifies to this solution's regular and mostly non-overlapping behavior.

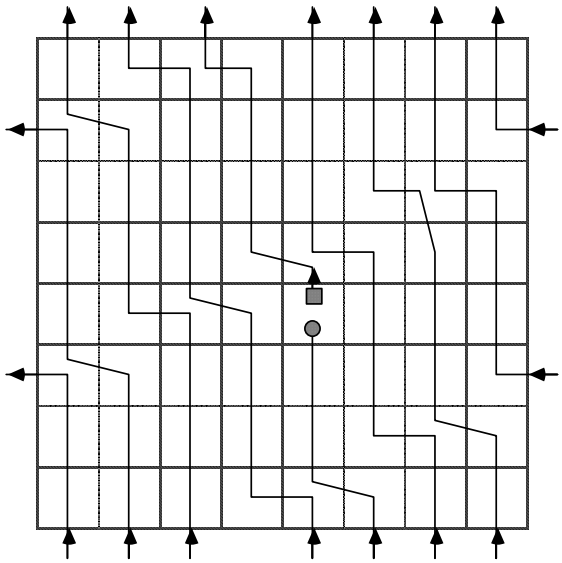

Figure 9 Trajectory of 42-point program with automatic function definition.

When this 42-point program is evaluated, ADF0 is executed first by the result-producing branch. ADF0 begins with a PROGN whose first argument is (V+ (#0,1) (#(2,0)). Since vector addition V+ has no side effects and since the return value of PROGN is the value returned

by its second argument, this first argument to the PROGN can be totally ignored. Since the remainder of ADF0 contains only (MOW) and (LEFT) operations, ADF0 returns (0,0). As it turns out, ADF1 never uses its argument.

The basic activity of ADF0 is to mow four squares of lawn in a northwesterly zigzag pattern. This zigzag action is illustrated at the starting point (3,3) in the middle of the figure. When simplified, ADF0 moves forward (i.e., north) one square and mows that square; it then turns left (i.e., west) and moves forward and mows that square; it then turns left three times (so that it is again oriented north); and it then moves and mows two squares.

The northwesterly zigzag mowing activity of ADF0 is then repeatedly invoked. The result-producing branch invokes ADF1 a total of four times. Each time ADF1 is invoked, ADF0 is invoked four times. This hierarchy of invocations produce a total of 16 calls for the zigzag activity of ADF0. Because of the initial direct call of ADF0 at the beginning of evaluation of the result-producing branch, the last of the 16 hierarchical invocations of ADF0 is not needed since the program is terminated by virtue of the completion of the overall task.

Note that this solution is an hierarchical decomposition of the problem. Genetic programming discovered a decomposition of the overall problem into 16 subproblems each consisting of the northwesterly zigzag mowing pattern. Genetic programming discovered the sequence of turns and moves to implement the northwesterly zigzag mowing activity. Genetic programming assembled the results of the northwesterly zigzag mowing into a solution of the overall problem by appropriately repositioning the lawn mower.

In a second run with automatic function definition, the following 100% correct 78-point program scoring 64 (out of 64) emerged in generation 2:

```
(progn (defun ADF0 ()
         (values (V+ (PROGN (V+ (V+ (LEFT) #(6 5))
           (PROGN (MOW) (LEFT))) (V+ (PROGN (MOW)
           (MOW)) (V+ (MOW) (MOW)))) (V+ (PROGN
           (V+ #(1 4) (MOW)) (PROGN #(3 1) (MOW)))
           (PROGN (PROGN #(3 1) (MOW)) (PROGN
           (LEFT) (LEFT)))))))
       (defun ADF1 (ARG0)
         (values (V+ (PROGN (FROG (PROGN ARG0
           (ADF0))) (V+ (PROGN (MOW) (ADF0)) (V+
           (V+ (ADF0) #(3 4)) (V+ (ADF0) ARG0))))
           (V+ (FROG (FROG (MOW))) (PROGN (PROGN
           (MOW) #(3 5)) (PROGN (MOW) (MOW)))))))
       (values (V+ (ADF1 (ADF1 (V+ #(7 1)
        (LEFT)))) (V+ (V+ (PROGN (LEFT) (LEFT))
        (V+ #(7 0) (LEFT))) (FROG (V+ (ADF0)
        (MOW)))))))
```

The result-producing branch of this 78-point program contains two invocations of ADF1, one invocation of ADF0, four (LEFT)'s, and one (MOW). ADF1 contains four invocations of ADF0, no turns, and five (MOW)'s. ADF0 contains eight (MOW)'s and four (LEFT)'s.

Figure 10 shows the trajectory of the lawn mower for this 78-point solution with automatic function definition. Again we see that the lawn mower here takes advantage of the regularity of the problem environment. Here it mows an entire row consisting of eight consecutive squares in an easterly direction and then proceeds to the next row to the south and does the same.

Note that this solution is an hierarchical decomposition of the problem. First, genetic programming discovered a decomposition of the overall problem into eight subproblems each consisting of mowing a single row of eight consecutive squares. Then, genetic programming discovered the sequence of turns and moves to implement the mowing of an entire row of eight squares. Thirdly, genetic programming assembled the results of the row mowing by repositioning the lawn mower to the next consecutive row.

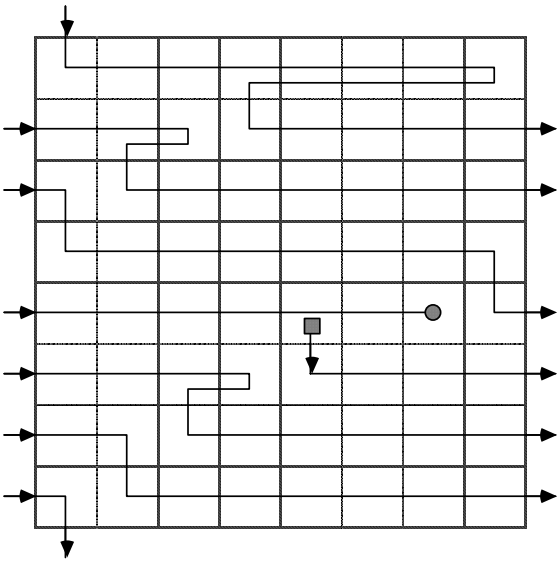

Figure 10 Trajectory of 78-point program with automatic function definition.

Over a series of 76 runs of this problem with automatic function definition, the average structural complexity of the 76 100%-correct solutions was 76.95 points. This average size is smaller by a factor of 3.65 than the average size of 280.82 without automatic function definition.

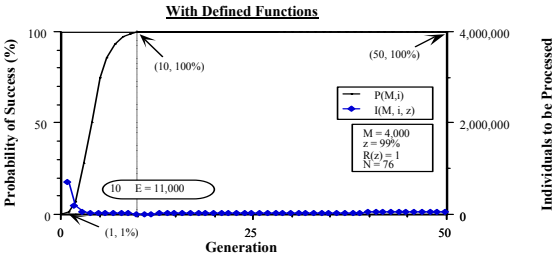

Figure 11 Performance curves showing that it is sufficient to process 11,000 individuals to yield a solution with 99% probability with automatic function definition.

Figure 11 presents the performance curves based on the 76 runs for this problem with automatic function definition. The cumulative probability of success $P(M,i)$ was 100% by generation 10. For a value of $P(M,i)$ of 100%, the number of independent runs, $R(z)$, necessary to yield a solution to the problem with a 99% probability by generation i is 1. The two numbers in the oval indicate that if this problem is run through to generation 10, processing a total of 11,000 individuals (i.e., 1,000 ∞ 11 generations ∞ 1 run) is sufficient to yield a solution to this problem with 99% probability.

9. CONCLUSION

This paper has described a general automatic approach for simultaneously discovering reusable subroutines and a way to invoke them to solve problems.

As we have now seen, genetic programming can solve this particular illustrative problem with or without automatic function definition.

Table 1 compares the solutions of this problem with and without automatic function definition with respect to the average structural complexity of the 100%-correct solutions and the computational effort $I(M,i,z)$ sufficient to yield a solution to this problem with 99% probability.

Table 1 Comparison table

	Without Automatic Function Definition	With Automatic Function Definition
Average Structural Complexity	280.82	76.95
Computational Effort $I(M,i,z)$	100,000	11,000

Table 1 shows that the average structural complexity, $\bar{S}$, of 280.82 points for 100% correct solutions without automatic function definition is 3.65 times the 76.95 points for such solutions with automatic function definition. That is, there is a reduction in the structural complexity of the solutions as a result of using automatic function definition.

Table 1 also shows that the 100,000 individuals required to be processed to yield an 100% correct solution to the problem with 99% probability without automatic function definition is 9.09 times the 11,000 individuals required with automatic function definition. That is, there is a reduction in the number of individuals required to be processed as a result of using automatic function definition.

Figure 10 summarizes these conclusions by showing that the structural complexity ratio is 3.65 and that the efficiency ratio is 9.09.

Figure 10 Summary graphs

ACKNOWLEDGEMENTS

James P. Rice of the Knowledge Systems Laboratory at Stanford University did the computer programming of the above on a Texas Instruments Explorer II+ computer.

REFERENCES

Davis, Lawrence (editor). *Genetic Algorithms and Simulated Annealing.* London: Pittman l987.

Davis, Lawrence. *Handbook of Genetic Algorithms*. New York: Van Nostrand Reinhold 1991.

Goldberg, David E. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley l989.

Holland, John H. *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: University of Michigan Press 1975. Revised Second Edition 1992 from The MIT Press.

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

Koza, John R. Hierarchical automatic function definition in genetic programming. In Whitley, Darrell (editor). *Proceedings of Workshop on the Foundations of Genetic Algorithms and Classifier Systems*, *Vail, Colorado 1992.* San Mateo, CA: Morgan Kaufmann Publishers Inc. 1992. 1992b.

Koza, John R. Simultaneous discovery of detectors and a way of using the detectors via genetic programming. *1993 IEEE International Conference on Neural Networks, San Francisco*. Piscataway, NJ: IEEE 1993. Volume III. Pages 1794-1801. 1993a.

Koza, John R. Simultaneous discovery of reusable detectors and subroutines using genetic programming. In Forrest, Stephanie (editor). *Proceedings of the Fifth International Conference on Genetic Algorithms*. San Mateo, CA: Morgan Kaufmann 1993b.

Koza, John R., Keane, Martin A., and Rice, James P. Performance improvement of machine learning via automatic discovery of facilitating functions as applied to a problem of symbolic system identification. *1993 IEEE International Conference on Neural Networks, San*

Francisco. Piscataway, NJ: IEEE 1993. Volume I. Pages 191-198. 1993 .

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

Michalewicz, Zbignlew. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag 1992.