Simultaneous Discovery of Reusable Detectors and 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 and then automatically discovering reusable subroutines, and a way of assembling the results produced by these subroutines in order to solve a problem. The approach uses genetic programming with automatic function definition. Genetic programming provides a way to genetically breed a computer program to solve a problem. Automatic function definition enables genetic programming to define potentially useful subroutines dynamically during a run. The approach is applied to an illustrative problem. Genetic programming with automatic function definition reduced the computational effort required to learn a solution to the problem by a factor of 2.0 as compared to genetic programming without automatic function definition. Similarly, the average structural complexity of the solution was reduced by about 21%.

1. INTRODUCTION AND OVERVIEW

An important goal of machine learning and artificial intelligence is the discovery of an automatic way to solve problems hierarchically.

The hierarchical approach to problem-solving can be viewed a three-step process. In the top-down way of describing this three-step process, one starts with the overall problem and seeks to discover a way to decompose the problem into subproblems. Second, one tries a way 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.

Solving some of the subproblems may require further invocation of this three-step process. If this three-step process is successful, one ends up with a hierarchical solution to the problem.

Hierarchical solutions to problems are potentially advantageous for machine learning because they avoid tediously re-solving what are essentially identical problems, because hierarchical solutions may be more parsimonious, and because hierarchical solutions may reduce the computational effort involved in doing the machine learning necessary to solve the problem.

The acceleration in learning is especially great when it is possible to reuse, with or without modification, the solutions to the subproblems. This acceleration is important because performance improvement by means of some kind of hierarchical approach appears to be necessary if machine learning methods are ever to be scaled up from small "proof of principle" problems to large problems.

Conventional approaches to machine learning usually require that the user hand-craft reusable subroutines for key features in the problem environment. Conventional approaches often additionally require the user to specify in advance the size and shape of the eventual way of combining the subroutines into a compete solution. However, in many instances, finding the reusable subroutines and a way of combining the subroutines in order to solve the problem really is *the problem*. Indeed, the necessity for pre-identification of the particular components of solutions and the necessity for pre-determination of a way of combining these components has been recognized as a bane of machine learning starting with Samuel's ground-breaking work in machine learning involving learning to play the game of checkers [Samuel 1959].

In Samuel's checkers player, learning consisted of progressively adjusting numerical coefficients in an algebraic expression of a predetermined functional form (specifically, a polynomial of specified order). Each component term of the polynomial represented a hand-crafted detector reflecting some aspect of the current state of the board (e.g., number of pieces, center control, etc.). The polynomial weighted each detector with a numerical coefficient and thereby assigned a single numerical value of a board to the player. If a polynomial were good at assigning values to boards, the polynomial could be used to compare the boards that would arise if the player were to make various alternative moves – thus permitting the best move to be selected from among the alternatives on the basis of the polynomial. In Samuel's learning system, the numerical coefficients of the polynomial were adjusted with experience, so that the predictive quality of the polynomial progressively improved. Samuel predetermined the way the detectors would be combined to solve the problem by selecting the functional form of the polynomial. Samuel recognized, from the beginning, the importance of enabling learning to occur without predetermining the size and shape of the solution and of

> "[getting] the program to generate its own parameters (detectors) for the evaluation polynomial."

This paper describes a general approach for simultaneously discovering reusable subroutines (detectors in Samuel's checker player) and a way of assembling calls to the reusable subroutines in order to solve a problem. Specifically, we will 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 automatic function definition to evolve a solution to the problem.

Genetic programming provides a way to search the space of all possible 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 useful functions automatically and dynamically during a run and also to combine these defined functions dynamically during a run in order to solve a problem.

Section 2 of this paper reviews genetic programming and section 3 describes automatic function definition. Section 4 states the illustrative problem. Section 5 details the preparatory steps for applying genetic programming to the problem. The problem is solved in section 6 without automatic function definition, and with automatic function definition in section 7. The two approaches are compared in section 8. Related and future work is discussed in section 9.

2. BACKGROUND

Since the invention of the genetic algorithm by John Holland [1975], the genetic algorithm has proven successful at finding an optimal point in a search space for a wide variety of problems.

Genetic programming is an extension of the genetic algorithm in which the genetic population consists of computer programs. Genetic programming provides a way to search the space of programs composed of certain terminals and primitive functions 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 (i.e., compositions of primitive functions and terminals) can be genetically bred to solve a surprising variety of problems in a wide variety of fields. A description of the crossover operation appropriate for programs is presented there in detail. A videotape visualization of numerous applications of genetic programming can be found in the *Genetic Programming: The Movie* [Koza and Rice 1992].

3. AUTOMATIC FUNCTION DEFINITION

When a human programmer writes a computer program to solve a problem, he often creates a subroutine (procedure, function) enabling a common calculation to be performed without tediously rewriting the code for that calculation.

For example, if a programmer needed to write a program for Boolean parity functions of several different orders, he might find it convenient first to write a subroutine for some lower-order parity function. He 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, if the programmer were using the LISP programming language, he might first write a function definition for the odd-2-parity function `xor` (exclusive-or) as follows:

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

This function definition (called a "`defun`" in LISP) does four things. First, it assigns a name, `xor`, to the function being defined thereby permitting subsequent reference to it. Second, 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 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, if the programmer needed the even-4-parity at some point in his main program, he might write

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

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

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, 1992b, 1993; 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 one or more of the defined functions. One defined function may hierarchically refer to another already-defined function (and potentially even itself), although such hierarchical or recursive references will not be used in this paper.

Figure 1 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.

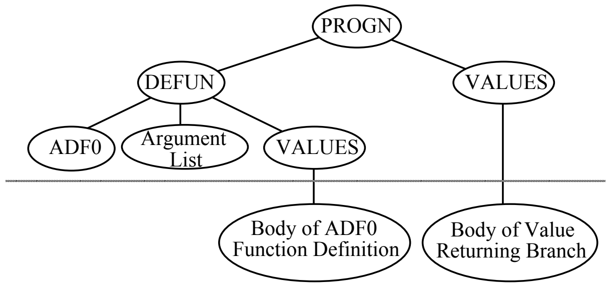
PROGN
DEFUN
VALUES
ADF0
Argument List
VALUES
Body of ADF0 Function Definition
Body of Value Returning Branch

Figure 1 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.

4. THE PROBLEM

After discovering that genetic programming with automatic function definition could solve Boolean parity problems of various orders [Koza 1992a, 1992b] as well as the discovery of an impulse response function of a time-invariant linear system [Koza, Keane, and Rice 1993], and to discovery of a pattern-recognizing program [Koza 1993], the question arose as to whether this new technique was applicable to other types of problems. This paper explores this question in the context of a problem requiring the discovery of 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.

The “San Mateo” trail consists of nine parts, each consisting of a square 13 by 13 grid containing different irregularities in the sequence of food. The irregularities include single and double gaps, corners where a single piece of food is missing, corners where there are two pieces of food that are missing in the trail’s current direction, and corners where there are two pieces of food that are missing to the left or right of the current direction of the trail.

Figure 2 shows the nine parts (i.e., fitness cases) of the San Mateo trail. Food is represented by solid black squares. The starting point of the ant within each part is

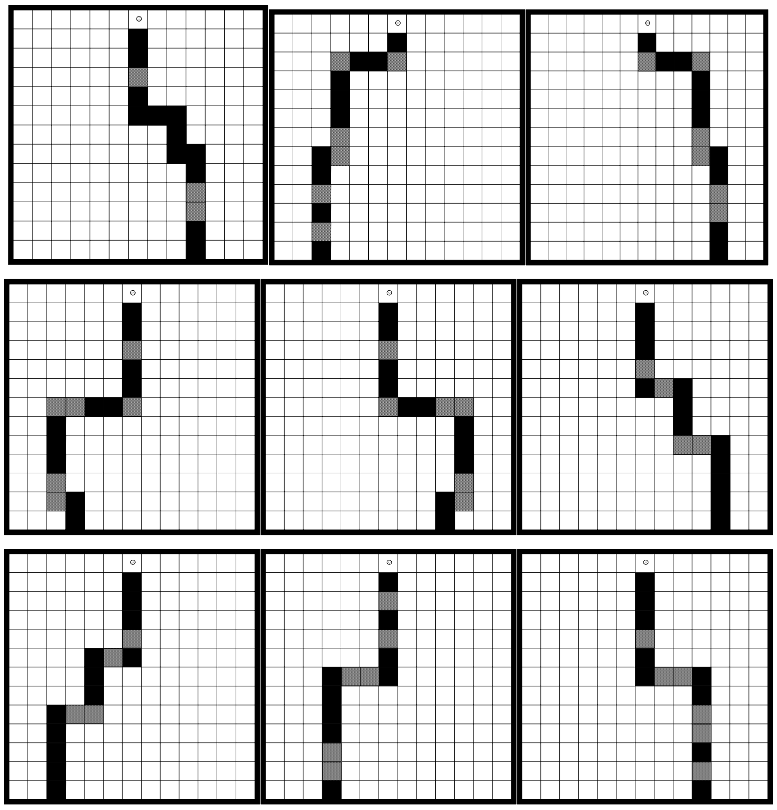

Figure 2 The 9 parts of the San Mateo trail

in the middle of the top row (denoted by a small circle in the figure). The ant faces south at the start of each part. There are a total of 96 pieces of food in the trail as a whole. For convenience of illustration, gaps in the trail are indicated by gray squares; however, the ant cannot distinguish between gray squares and white squares.

In the first of the nine parts of the trail, there are 12 pieces of food and the irregularities in the trail consist only of one single gap and one double gap. The last four parts of the trail each contain an instance of the most difficult irregularity, namely corners where two pieces of food are missing to the left or right of the current direction of the trail.

The original version of this problem involving a simpler Santa Fe trail was solved in Jefferson et al. (1991).

5. PREPARATORY STEPS FOR GENETIC PROGRAMMING WITHOUT AUTOMATIC FUNCTION DEFINITION

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

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

The terminal set $\mathcal{T}$ for this problem consists of

$$\mathcal{T} = \{\texttt{(RIGHT)}, \texttt{(LEFT)}, \texttt{(MOVE)}\},$$

`(RIGHT)`, `(LEFT)`, and `(MOVE)` are each operators that take no explicit arguments, but have side effects on the state of the ant.

`(RIGHT)` turns the facing direction of the ant right by 90° (without moving the ant).

`(LEFT)` turns the facing direction of the ant left by 90° (without moving the ant).

`(MOVE)` moves the ant forward in the direction it is currently facing. When an ant moves into a square, it eats the food, if there is any, in that square (thereby removing that piece food from that square). Moreover, the eating of a piece of food throws execution of the program back to its beginning.

The function set $\mathcal{F}$ consists of

$$\mathcal{F} = \{\texttt{IF-FOOD-AHEAD}, \texttt{PROGN}\},$$

with these functions each taking 2 arguments.

`IF-FOOD-AHEAD` permits the ant to sense the single adjacent square in the direction the ant is currently facing. This conditional branching operator takes two arguments and executes the first argument if (and only if) if there is currently food in the single adjacent square in the direction the ant is currently facing, but executes the second argument if (and only if) if there is currently no food in that square. This conditional branching operator is implemented as a macro as described in Koza [1992a].

`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 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 ant's goal is to eat as much food as possible in the nine parts of the overall San Mateo trail. Each individual in the population is tested against an environment consisting of $N_{fc} = 9$ fitness cases, each consisting of one of the parts of the San Mateo trail. The raw fitness of a particular program is the number of pieces of food (from 0 to 96) eaten over the nine parts of the trail.

The movement of the ant is terminated on a particular part of the trail when the ant touches the outer boundary of the

13 by 13 grid or it has executed a total of 120 `RIGHT` or `LEFT` turns or 80 `MOVE`s for the current part of the trail. The amount of food eaten up to the time of termination on each part of the trail is accumulated over the nine parts of the trail.

Standardized fitness is the total amount of available food (i.e., 96) minus raw fitness.

The fourth major step in preparing to use genetic programming is the selection of values for certain parameters. Our choice of 4,000 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 reproduction).

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 a 100% correct individual 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 successful run of genetic programming without automatic function definition on this problem, the following 95-point individual collecting 96 (out of 96) pieces of food emerged on generation 13:

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

Over a series of 26 runs of this problem with a population of 4,000 without automatic function definition, the average structural complexity (i.e., functions and terminals in the program) of the 22 100%-correct solutions was 90.9 points.

The rising curve in figure 3 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 96). As can be seen, the experimentally observed value of $P(M,i)$ is 69% by generation 16, and 85% by generation 50 over the 26 runs.

The second curve in figure 3 (which first falls and then rises) 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 brackets indicate the ceiling function for rounding up to the next highest integer. Throughout this paper, the probability z will be 99%.

The $I(M,i,z)$ curve reaches a minimum value at generation 16 (highlighted by the light dotted vertical line). For a value of $P(M,i)$ of 69%, the number of independent runs, $R(z)$, necessary to yield a solution to the problem with a 99% probability by generation i is 4. The two summary numbers (16 and 272,000) in the oval indicate that if this problem is run through to generation 16 (the initial random generation being counted as generation 0), processing a total of 272,000 individuals (i.e., 4,000 ∞ 17 generations ∞ 4 runs) is sufficient to yield a solution to this problem with 99% probability. This number, 272,000, is a measure of the computational effort necessary to yield a solution to this problem with 99% probability *without automatic function definition*.

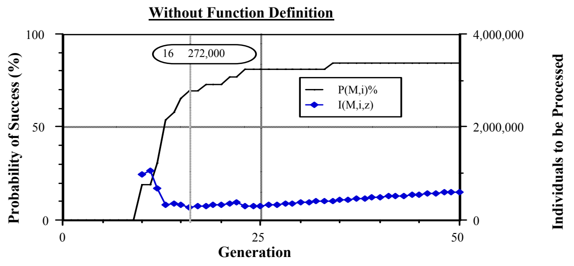

Figure 3 Performance curves showing that it is sufficient to process 272,000 individuals to yield a solution with 99% probability without automatic function definition.

7. PREPARATORY STEPS FOR GENETIC PROGRAMMING WITH

AUTOMATIC FUNCTION DEFINITION

In applying genetic programming with automatic function definition to this problem, we decided that each individual program in the population will consist of one function-defining branch defining a function called `ADF0` taking no arguments and one result-producing branch.

We first consider the function-defining branch.

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

$$\mathcal{T}_{fd} = \{\texttt{(RIGHT), (LEFT), (MOVE)}\}.$$

The function set $\mathcal{F}_{fd}$ for `ADF0` is

$$\mathcal{F}_{fd} = \{\texttt{IF-FOOD-AHEAD, PROGN}\},$$

each taking 2 arguments.

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

We now consider the result-producing branch.

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

$$\mathcal{T}_{rp} = \{\texttt{(RIGHT), (LEFT), (MOVE)}\}.$$

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

$$\mathcal{F}_{rp} = \{\texttt{ADF0, IF-FOOD-AHEAD, PROGN}\},$$

with the functions taking 0, 2, 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 one function-defining branch and one result-producing branch, we must create the initial random generation so that every individual program in the population has this particular constrained syntactic structure. Specifically, every individual program must have the invariant structure represented by the six points of types 1 through 6 described above. Each function and terminal in the function-defining branch is of type 7. Each function and terminal in the result-producing branch is of type 8.

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. Since each program must have the invariant structure represented by the six points of types 1 through 6, crossover is limited to points of types 7 and 8. Structure-preserving crossover is implemented by limiting crossover to points of type 7 or 8. This restriction on the selection of the crossover point of the second parent ensures the offspring's syntactic validity.

Genetic programming will evolve a different function definition in the function-defining branch of each overall program and then, at its discretion, it may call the defined function 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 successful run of genetic programming with automatic function definition on this problem, about half (2,044 of the 4,000) of the individuals in generation 0 scored zero in their search for food over the 9 parts of the San Mateo trail. Most of these individuals turned and looked, but were immobile. Another 20% (868) scored 18 out of 96 because there are, over the 9 parts of the trail, 18 pieces of food available to a program that merely moves south whenever food is present to the south. About 1% of the individuals scored between 54 and 72.

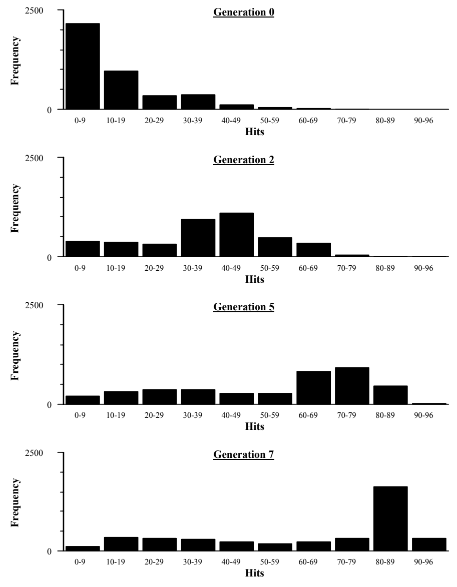

Figure 4 Hits histograms

Figure 4 shows the hits histograms for generations 0, 2, 5, and 7 of this run. Notice the left-to-right undulating movement of both the high point and the center of mass of these histograms. This "slinky" movement reflects the improvement of the population as a whole.

In generation 7 of this run, the following 100% correct solution emerged:

```
(progn (defun ADF0 ()
            (values (PROGN (IF-FOOD-AHEAD (IF-FOOD-AHEAD
                 (MOVE) (RIGHT)) (PROGN (LEFT) (MOVE))) (PROGN
                 (IF-FOOD-AHEAD (IF-FOOD-AHEAD (MOVE) (LEFT))
        (PROGN (PROGN (RIGHT) (LEFT)) (PROGN (LEFT)
        (MOVE)))) (IF-FOOD-AHEAD (LEFT) (RIGHT))))
      (values (PROGN (PROGN (MOVE) (ADF0)) (PROGN (IF-
        FOOD-AHEAD (MOVE) (MOVE)) (PROGN (ADF0)
        (ADF0))))).
```

In this program, ADF0 is invoked three times from the result-producing branch. The result-producing branch serves to reposition the ant just prior to the first and second invocation of ADF0.

Figure 5 shows the trajectory for the ninth fitness case of the ant for this run. In this figure (and the following figures in this section), the light lines represent movements executed while in the result-producing branch of the program, while the heavy lines indicate movements executed by the automatically defined function ADF0. The figure is suggestive of the reuse of a semicircular counterclockwise inspecting motion.

The best-of-run individual from generation 7 can be simplified as follows:

```
(progn (defun ADF0 ()
        (values (IF-FOOD-AHEAD (MOVE)                    ;a
                    (PROGN (LEFT) (MOVE)                 ;b
                       (IF-FOOD-AHEAD (MOVE)             ;c
                          (PROGN (LEFT) (MOVE)           ;d
                             (IF-FOOD-AHEAD (LEFT)
                                (RIGHT))))))))
        (values (PROGN (MOVE) (ADF0) (MOVE) (ADF0)
                   (ADF0))))
        ;           R     1     P     2     Q
```

Figure 6 shows the trajectory of the artificial ant executing this semicircular counterclockwise inspecting motion specified by the best-of-run individual from generation 7. For simplicity, this figure shows only part of the 13 by 13 grid and contains food in only four squares. As usual, the ant starts at the circle in the top row.

Since the ant encounters food on each of its first four downward movements, evaluation of the program terminates upon execution of the first (MOVE) operation (labeled "1") in the result-producing branch. The four places on the trajectory where this occurs are similarly labeled "1."

The remainder of the trajectory shown represents three evaluations of the program. These three executions occur in the absence of any food. Each circle denotes the ant's exit from one invocation of ADF0. The two filled circles (labeled "E") denote the ant's exit from the first and second of the three evaluations of the program. The large filled circle denotes the ant's exit from the third evaluation of the program.

Lines in the figure labeled with "2" denote a movement caused by the second (MOVE) operation of the result-producing branch.

Points in the figure labeled with capital letters (P, Q, or R) denote invocations of ADF0 by the result-producing branch.

All the bold lines in the figure denote a movement caused by the (MOVE) operations on lines "b" or "d" of ADF0.

Note that this solution is a hierarchical decomposition of the problem. First, genetic programming discovered a decomposition of the overall problem into a subroutine for performing an inspecting motion. Then, genetic programming discovered the sequence of sensor tests, turns, and moves to implement this inspecting motion. Thirdly, genetic programming assembled the results of three such inspecting motions along with other stage-setting sensor tests, turns, and moves into a solution of the overall problem.

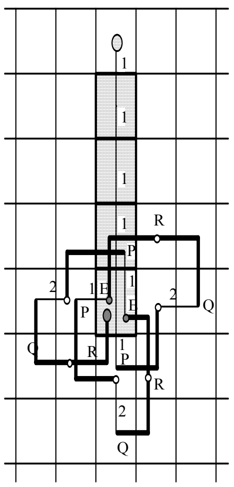

Figure 6 Trajectory of artificial ant showing its semicircular counterclockwise inspecting motion.

Over a series of 19 runs of this problem with automatic function definition, the average structural complexity of the 19 100%-correct solutions was 71.7 points. This average size is smaller than the average size of 90.9 when automatic function definition is not used.

Figure 7 presents the performance curves based on the 19 runs for this problem with automatic function definition. The cumulative probability of success $P(M,i)$ was 95% by generation 16 and was 100% by generation 33. The two numbers in the oval indicate that if this problem is run through to generation 16, processing a total of 136,000 individuals (i.e., 4,000 ∞ 17 generations

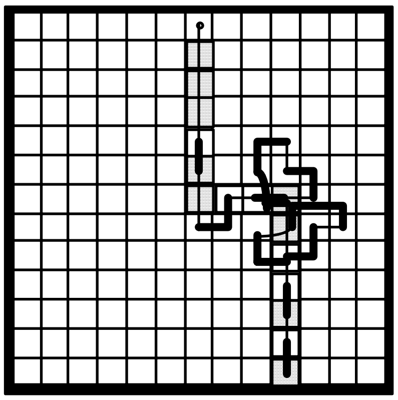

Figure 5 Trajectory of artificial ant for the ninth fitness case of the successful run solving on generation 7.

∞ 2 runs) is sufficient to yield a solution to this problem with 99% probability.

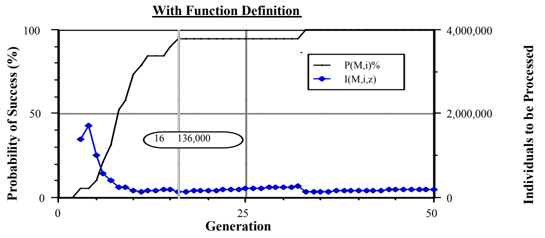

Figure 7 Performance curves showing that it is sufficient to process 136,000 individuals to yield a solution with 99% probability with automatic function definition.

The 136,000 individuals that must be processed to yield a solution with automatic function definition is half of the 272,000 individuals required when automatic function definition is not used. In particular, automatic function definition is 2.0 times more efficient.

9. CONCLUSION

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

As we have now seen, genetic programming can solve a 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 $\bar{S}$	90.9	71.7
Computational effort - I(M,i,z)	272,000	136,000

As can be seen from table 1, there is a reduction in the structural complexity of the solutions as a reult of using automatic function definition. The ratio of the average structural complexity, $\bar{S}$, between the two approaches is 1.27. In addition, there is a reduction in the computational effort required to solve the problem when using automatic function definition. The ratio of the computational effort between the two approaches is 2.00.

Figure 8 compares the information in table 1 showing these two ratios.

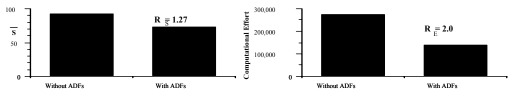

Figure 8 Summary graphs

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

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.

Jefferson, David, Collins, Robert, Cooper, Claus, Dyer, Michael, Flowers, Margot, Korf, Richard, Taylor, Charles, and Wang, Alan. Evolution as a theme in artificial life: The genesys/tracker system. In Langton, Christopher, Taylor, Charles, Farmer, J. Doyne, and Rasmussen, Steen (editors). *Artificial Life II, SFI Studies in the Sciences of Complexity*. Volume X. Redwood City, CA: Addison-Wesley 1991. Pages 549-578.

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. 1993.

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

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 .

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