John R. Koza

This chapter uses three differently sized versions of an illustrative problem that has considerable regularity, symmetry, and homogeneity in its problem environment to compare genetic programming with and without the newly developed mechanism of automatic function definition. Genetic programming with automatic function definition can automatically decompose a problem into simpler subproblems, solve the subproblems, and assemble the solutions to the subproblems into a solution to the original overall problem. The solutions to the problem produced by genetic programming with automatic function definition are more parsimonious than those produced without it. Genetic programming requires fewer fitness evaluations to yield a solution to the problem with 99% probability with automatic function definition than without it.

When we consider the three differently sized versions of the problem we find that the size of the solutions produced *without* automatic function definition can be expressed as a direct multiple of problem size. In contrast, the average size of solutions *with* automatic function definition is expressed as a certain minimum size representing the overhead associated with automatic function definition; however, there is only a very slight increase in the average size of the solutions with problem size. Moreover, the number of fitness evaluations required to yield a solution to the problem with a 99% probability grows very rapidly with problem size without automatic function definition, but this same measure grows only linearly with problem size with automatic function definition.

5.1 Introduction

Hierarchical problem-solving ("divide and conquer") may be advantageous in solving large and complex problems because the solution to an overall problem may be found by decomposing it into smaller and more tractable subproblems in such a way that the solutions to the subproblems are reused many times in assembling the solution to the overall problem.

In the top-down way of describing this three-step hierarchical problem-solving process, one first tries to discover a way to decompose a given problem into subproblems (modules). 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. Presumably, solving the subproblems will prove to be simpler than solving the original overall problem. In practice, solving some of the subproblems may lead to a recursive reinvocation of this three-step process. In any event, if this three-step process is successful, one ends up with a hierarchical (modular) solution to the problem.

Hierarchical solutions to problems are potentially very favorable because they avoid tediously re-solving what are essentially identical subproblems. Consequently, such

solutions may be more parsimonious. They may also require less total effort to solve the overall problem. Leverage gained from the successful reuse of solutions to subproblems 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.

In the terminology of computer programming, the three-step process of solving problems hierarchically starts by analyzing the overall problem and dividing it into parts. Second, one writes subprograms (subroutines, procedures, defined functions) to solve each part of the problem. Third, one writes a main program or other calling program that invokes the subprograms and assembles the results produced by the functions into a solution to the overall problem. In practice, the result produced by a computation can be a single value, a set of values, the side-effects performed on a system, or a combination thereof.

This same three-step process of solving problems hierarchically can also be described in a bottom-up way. First, one seeks to discover regularities and patterns at the lowest level of the problem environment. Second, one restates or recodes the problem in terms of these regularities so as to create a new problem stated in new terms. Third, one tries to discover a solution to the presumably simpler recoded problem. In practice, the process of discovering a solution to the recoded problem may recursively involve further discovery of regularities and patterns and further recoding. If this process of regularity finding and assembly is successful, one ends up with a hierarchical solution to the problem.

The bottom-up way of solving problems hierarchically is often viewed as a change of representation. The recoding of the original problem is a change of representation from the original representation of the problem to a higher level. New regularities often become apparent when the representation is changed in this way.

The goal of automatically solving problems hierarchically (whether top-down or bottom-up) has been a central issue in machine learning and artificial intelligence since the beginning of these fields.

The one obvious question concerning this alluring three-step process of solving problems hierarchically is how does one go about implementing this process in an automated and domain independent way? The discovery of a solution to a subproblem (i.e., the second step of the top-down approach) can potentially be accomplished by means of genetic programming. Indeed, The book *Genetic Programming: On the Programming of Computers by Means of Natural Selection* [Koza 1992a] demonstrated that a variety of problems can be solved, or approximately solved, by genetically

breeding a population of computer programs over a period of many generations. See also [Koza and Rice 1992].

But what about the other steps of this three-step problem-solving process? This chapter illustrates, for a particular problem, that when the recently developed mechanism of automatic function definition [Koza 1992a, 1992b, 1993; Koza, Keane, and Rice 1993] is added to genetic programming, *all three steps* in the hierarchical problem-solving process described above can be simultaneously performed within a single run of genetic programming. An *automatically defined function* (an *ADF*) is a function (i.e., subroutine, procedure, module) that is evolved during a run of genetic programming and which may be called by the main program (or other calling program) that is being simultaneously evolved during the same run.

5.2 The Lawn Mower Problem

In the lawn mower problem, the goal is to find a program for controlling the movement of a lawn mower so that the lawn mower cuts all the grass in the yard.

We first consider a version of this problem in which the lawn mower operates in an 8 by 8 toroidal square area of lawn that initially has grass in all 64 squares.

Each square of the lawn is uniquely identified by a vector of integers modulo 8 of the form (i,j), where $0 \leq i, j \leq 7$. The lawn mower starts at location (4,4) facing north. The state of the lawn mower consists of its location on one of the 64 squares of the lawn and the direction in which it is facing. The lawn is toroidal in all four directions, so that whenever the lawn mower moves off the edge of the lawn, it reappears on the opposite side. There are no obstacles in the yard.

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 all the grass, if any, in the square onto which it arrives. The lawn mower has no sensors.

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 (and appropriate turning actions). Instead, a human programmer would exploit the considerable regularity, symmetry and homogeneity inherent in this problem environment by writing a program that mows a certain small area of the lawn in a particular way, then repositions the lawn mower in some regular way, and then repeats 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 a small area), solve the subproblem, and then repeatedly reuse the subproblem solution at different places on the lawn in order to solve the overall problem.

5.3 Preparatory Steps Without Automatic Function Definition

The operations of turning left, moving one square and then mowing, and jumping and then mowing each change the state of the lawn mower and are side-effecting operators that take no arguments. They can be treated as terminals.

Since it may be desirable to be able to manipulate the numerical location of the lawn mower using arithmetic operations, both random constants and arithmetic operations should be available as ingredients of programs for solving this problem. The terminal set should thus include random constants. The random constants, ←, appropriate for this problem are vectors `(i,j)` of integers modulo 8.

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

$\mathcal{T}$= {`(LEFT)`, `(MOW)`, ←}.

The operator `LEFT` takes no arguments and turns the orientation of the lawn mower counter-clockwise by 90° (without moving the lawn mower). Since the programs will be performing arithmetic, it is necessary that all terminals and functions return a value that can serve as a legitimate argument to the arithmetic operations. Thus, to assure closure, `LEFT` returns the vector value `(0,0)`.

The operator `MOW` takes no arguments and moves the lawn mower in the direction it is currently facing and mows the grass, if any, in the square to which it is moving (thereby removing all the grass, if any, from that square). `MOW` does not change the facing direction of the lawn mower. For example, if the lawn mower is at location (1,3) and 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) and facing east, `MOW` moves the lawn mower to location (0,3) because of the toroidal geometry. To assure closure, `MOW` also returns the vector value `(0,0)`.

The function set consists of

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

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

V+ is two-argument vector addition function modulo 8. For example, (V+ (1,2) (3,7)) returns the 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 and mows the grass, if any, in the square on which the lawn mower arrives (thereby removing all the grass, if any, from that square). FROG does not change the facing direction of the lawn mower. 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. Thus, (FROG (3,5)) returns the value (3,5).

The problem can be solved with either the MOW or FROG operator; however, we include both operators to enrich the function set by allowing alternatives approaches for solving the problem.

The goal is to mow all 64 squares of grass. The movement of the lawn mower is terminated when either the lawn mower has executed a total of 100 LEFT turns or a total of 100 movement-causing operations (i.e., MOWs or FROGs). The raw fitness of a particular program is the amount of grass (from 0 to 64) mowed within this allowed amount of time. Since the yard contains no obstacles and the toroidal topology of the yard is perfectly symmetrical, it is only necessary to measure fitness over one fitness case for this problem.

Table 5.1 summarizes the key features of the lawn mower problem with 64 squares. The last seven rows of this table apply to automatic function definition (described below).

5.4 Lawn Size of 64 Without Automatic Function Definition

The only way to write a computer program to mow all 64 squares of the lawn with the available movement-causing and turning operators involves tediously writing a program consisting of at least 64 MOWs or FROGs so that all 64 squares of the lawn are mowed. One possible orderly way of writing this tedious program involves mowing all eight squares of lawn in the vertical column beginning at the starting location (4,4), turning left upon returning to (4,4), moving and mowing one column to the west, turning left three times so as to face north again, and mowing all eight squares of lawn in the new vertical column.

Subtrees of somewhat effective programs in this problem typically mow small portions of the lawn. If two programs are selected from the population based on their fitness, both of the selected programs will usually mow an above-average amount of lawn for their

Tableau for the lawn mower problem with 64 squares.

Objective:	Find a program to control a lawn mower so that it mows the grass on all 64 squares of lawn in an unobstructed yard.
Terminal set without automatic function definition:	`(LEFT)`, `(MOW)`, ←.
Function set without automatic function definition:	`V+`, `FROG`, `PROGN`.
Fitness cases:	One fitness case consisting of a toroidal lawn with 64 squares, each initially containing grass.
Raw fitness:	Raw fitness is the amount of grass (from 0 to 64) mowed within the maximum allowed number of state-changing operations.
Standardized fitness:	Standardized fitness is the total number of squares (i.e., 64) minus raw fitness.
Hits:	Same as raw fitness.
Wrapper:	None.
Parameters:	M = 1,000. G = 51.
Success predicate:	A program scores the maximum number of hits.
Overall program structure with automatic function definition:	One result-producing branch and two function definitions with `ADF0` taking no arguments and `ADF1` taking one argument `ARG0`.
Terminal set for the result-producing branch:	`(LEFT)`, `(MOW)`, ←.
Function set for the result-producing branch:	`ADF0`, `ADF1`, `V+`, `FROG`, `PROGN`.
Terminal set for the function definition `ADF0`	`(LEFT)`, `(MOW)`, ←.
Function set for the function definition `ADF0`	`V+`, `PROGN`.
Terminal set for the function definition `ADF1`	`ARG0`, `(LEFT)`, `(MOW)`, ←.
Function set for the function definition `ADF1`	`ADF0`, `V+`, `FROG`, `PROGN`.

generation. Moreover, a random subtree from either of these selected individuals will, on average, mow an above-average amount of lawn for its generation. Thus, the effect of the crossover operation is to create new programs which will, on average, mow an increasing and above-average amount of lawn.

Table 5.1

The following 296-point individual achieving a raw fitness of 64 emerged on generation 34 of this run without automatic function definition:

```
(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)))
```

This 296-point program solves the problem by agglomerating enough erratic movements so as to cover the entire area of the lawn within the allowed maximum number of operations. In fact, the way that this program solves the problem is so tedious and convoluted that it can be easily visualized only after dividing the trajectory of the lawn mower into three epochs.

Figure 5.1 shows a partial trajectory of this best-of run 296-point individual for a first epoch consisting of mowing operations 0 through 30 for the lawn mower problem; figure 5.2 shows a partial trajectory for a second epoch consisting of mowing operations 31 through 60; and figure 5.3 shows a partial trajectory for a third epoch consisting of mowing operations 61 through 85. As can be seen, even though the problem environment contains considerable regularity, symmetry, and homogeneity in that it requires mowing all 64 squares in an unobstructed toroidal yard, this solution operates in an entirely *ad hoc* fashion. For example, between mowing operations 2 and 3, the lawn mower `FROG`s up two rows and three columns to the right; between operations 4 and 5, the mower `FROG`s up six rows and three columns to the left; and between operations 6 and 7, the mower `FROG`s up two (i.e., down six) and two columns to the right.

Over 38 runs, the average structural complexity (i.e., total number of functions and terminals in the program) of the 35 successful solutions to the lawn mower problem without automatic function definition was 280.82 points. The structural complexity of the successful solutions is about 4.4 times the size of the lawn. The successful programs are so large because they make no use of the inherent regularity of the problem environment.

Using methods described in detail in [Koza 1992a], we find that the total number of individuals $I(M,i,z)$ that need to be processed in order to solve this problem with a 99% probability is 100,000.

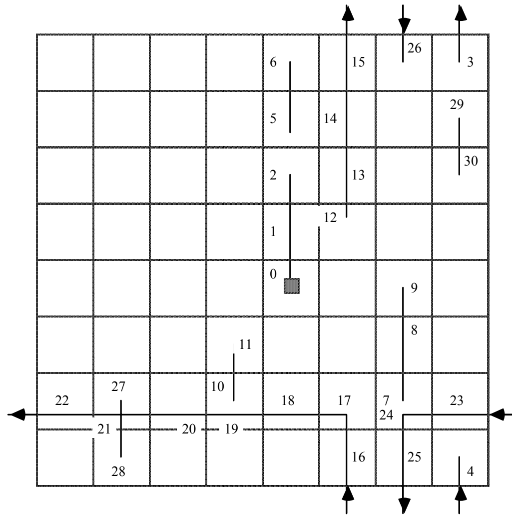

Figure 5.1
First partial trajectory of 296-point program for mowing operations 0 through 30 without automatic function definition.

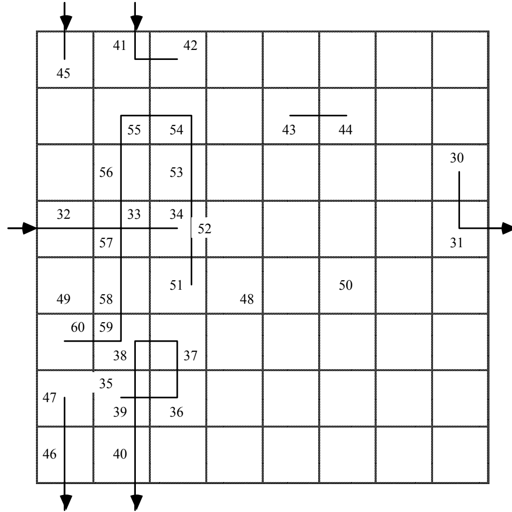

Figure 5.2
Second partial trajectory of 296-point program for mowing operations 31 through 60 without automatic function definition.

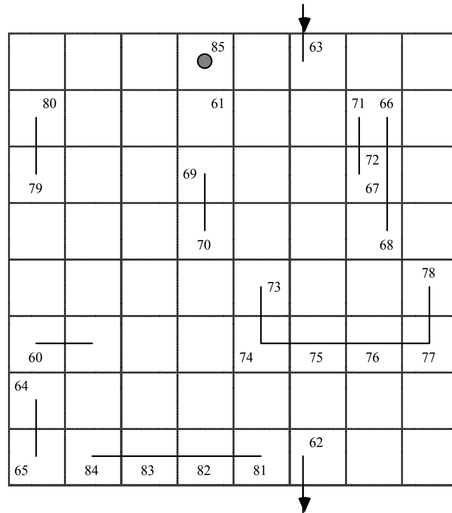

Figure 5.3
Third partial trajectory of 296-point program for mowing operations 61 through 85 without automatic function definition.

5.4.1 Lawn Size of 96 Without Automatic Function Definition

When the size of the problem is scaled up by 50% from 64 to 96 squares of lawn, the average size of the successful solutions among 197 runs increases to 426.9 (i.e., about 4.4 times the lawn size). The number of fitness evaluations required to yield a solution of the problem with 99% probability increases substantially to 4,539,000.

5.4.2 Lawn Size of 32 Without Automatic Function Definition

When the size of the problem is scaled down by 50% from 64 to 32 squares of lawn, the average size of the successful solutions among 64 runs decreases to 145 (i.e., about 4.5 times the lawn size). The number of fitness evaluations required to yield a solution of the problem with 99% probability decreases substantially to 19,000.

5.5 Preparatory Steps With Automatic Function Definition

Each of the programs presented in the previous section for solving the lawn mower problem without automatic function definition contained at least 64 `MOW`s or `FROG`s

when the lawn size is 64. However, a human programmer would never consider solving this problem in this tedious way. Instead, a human programmer would write a program that first mows a certain small sub-area of the lawn in some orderly way, then repositions the lawn mower to a new sub-area of the lawn in some orderly (probably tessellating) way, and then repeats the mowing action on the new sub-area of the lawn. The program would contain a sufficient number of invocations of the orderly method for mowing sub-areas of the lawn so as to mow the entire lawn. That is, a human programmer would exploit the considerable regularity and symmetry inherent in the problem environment by decomposing the problem into subproblems and then would repeatedly reuse the solution to the subproblem in order to solve the overall problem.

In applying genetic programming with automatic function definition to the lawn mower problem, we 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. The second defined function `ADF1` can hierarchically refer to the first defined function `ADF0`.

We first consider the two function-defining branches.

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

$\mathcal{T}_{fd0}$ = {`(LEFT)`, `(MOW)`, ←}.

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

$\mathcal{F}_{fd0}$ = {`V+`, `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 $\mathcal{T}_{fd0}$.

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

$\mathcal{T}_{fd1}$ = {`ARG0`, `(LEFT)`, `(MOW)`, ←}.

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

$\mathcal{F}_{fd1}$ = {`ADF0`, `V+`, `FROG`, `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, the value returned by ADF0 and ADF1 is either (0,0) or the result of the vector addition V+ operating on random constants, or on ARG0, the value of calls to ADF0 and random constants in the case of ADF1.

We now consider the result-producing branch.

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

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

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

$\mathcal{F}_{rp}$ = {ADF0, ADF1, V+, FROG, 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 $\mathcal{T}_{rp}$.

The last seven rows of table 5.1 summarize the key features of this problem with automatic function definition.

Additional details about automatic function definition and the structure-preserving crossover required with automatic function definition can be found in chapter 2 of this book, the forthcoming book *Genetic Programming II* [Koza 1994], and the forthcoming videotape *Genetic Programming II Videotape: The Next Generation* [Koza and Rice 1994].

5.6 Lawn Size of 64 With Automatic Function Definition

When genetic programming with automatic function definition is applied to this problem, the results are very different from the haphazard solution obtained without automatic function definition.

In one run of this problem 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, genetic programming discovered a decomposition of the overall problem into five subproblems (four `ADF1`'s and one `ADF0`) in the result-producing branch at the top level. The result-producing branch does not contain any `LEFT`, `MOW`, or `FROG` operations at all. `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 5.4 shows the trajectory of the lawn mower for this 42-point solution. Note the difference between this regular trajectory and the haphazard character of the three partial trajectories shown in figures 5.1, 5.2, and 5.3. The lawn mower here takes advantage of the regularity, symmetry, and homogeneity of the problem environment. It performs a tessellating activity that covers the entire lawn. 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 behavior.

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 dummy variable.

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 (4,4) in the middle of the figure. `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

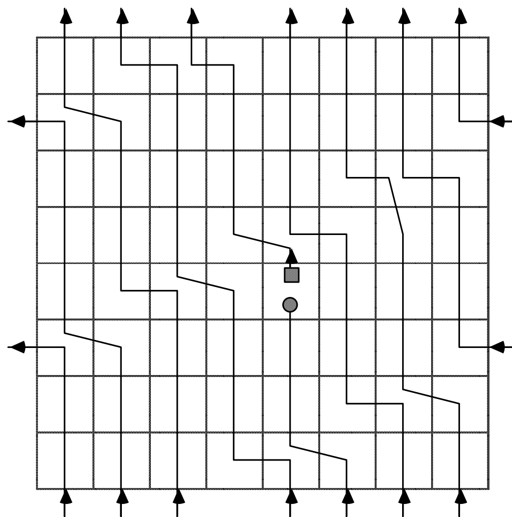

Figure 5.4
Trajectory of lawn mower using 42-point program with automatic function definition.

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 produces a total of 16 calls for the zigzag activity of ADF0. Because of the initial direct call of ADF0 at the beginning of the 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.

This zigzagging solution is an hierarchical decomposition and solution of the problem involving three simultaneous, automatic discoveries. First, genetic programming discovered a decomposition of the overall problem into 16 subproblems each consisting of the northwesterly zigzag mowing pattern. Second, genetic programming discovered the sequence of turns and moves to implement the northwesterly zigzag mowing activity. Third, genetic programming assembled the results of this mowing motion into a solution of the overall problem by appropriately repositioning the lawn mower.

In other runs, some 100% correct solutions mowed entire rows and columns while others zigzagged, swirled, crisscrossed, and leapfrogged around the lawn in a tessellating manner.

The average structural complexity of the 76 successful solutions to the lawn mower problem with automatic function definition was 76.95 points.

11,000 fitness evaluations are required to yield a solution to the lawn mower problem with 64 squares of lawn with automatic function definition with 99% probability.

The beneficial effect of automatic function definition can be seen by taking the ratio of the average structural complexities (280.82 and 76.95) for the successful solutions to this problem without and with automatic function definition. This ratio (3.65 here) is called the structural complexity ratio and provides a way of measuring the parsimony (or lack of parsimony) produced by automatic function definition. Similarly, the ratio of the minimal $I(M,i,z)$ numbers (100,000 and 11,000), called the efficiency ratio, provides a way of measuring the improvement (or degradation) in computation effect produced by automatic function definition.

Figure 5.5 shows that the structural complexity ratio for the lawn mower problem with 64 squares is 3.65 and the efficiency ratio is 9.09. Since both ratios are above 1.00, automatic function definition improves both parsimony and efficiency.

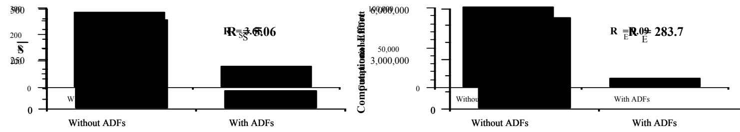

Figure 5.5
Summary graphs for the lawn mower problem with 64 squares.
Figure 5.6
Summary graphs for the lawn mower problem with 96 squares.

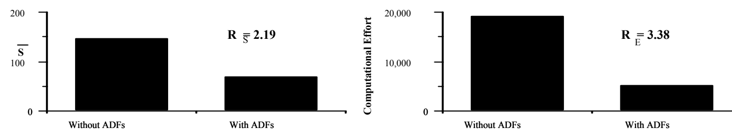

Figure 5.7
Summary graphs for the lawn mower problem with 32 squares.

5.6.1 Lawn Size of 96 With Automatic Function Definition

When the size of the problem is scaled up from 64 to 96 squares, the average structural complexity of the successful solutions to the lawn mower problem over 52 runs with automatic function definition was 84.3 points.

The number of fitness evaluations required to yield a solution to the problem with 99% probability with automatic function definition does not increase in the same dramatic way as is the case without automatic function definition. In fact, only 16,000 fitness evaluations are required.

Figure 5.6 shows that the structural complexity ratio for the lawn mower problem with 96 squares is 5.06 and the efficiency ratio is 283.7.

5.6.2 Lawn Size of 32 With Automatic Function Definition

When the size of the problem is scaled down from 64 to 32 squares, the average structural complexity of the successful solutions to the lawn mower problem over 52 runs with automatic function definition was 66.3 points.

The number of fitness evaluations required to yield a solution to the problem with 99% probability with automatic function definition is 5,000.

Figure 5.7 shows that the structural complexity ratio for the lawn mower problem with 32 squares is 2.19 and the efficiency ratio is 3.8.

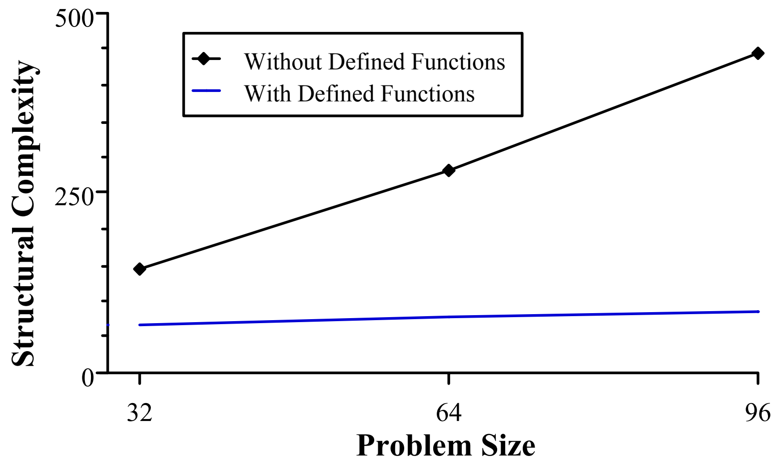

Figure 5.8
Comparison of average structural complexity of the successful programs for lawn sizes of 32, 64, and 96 both with and without automatic function definition.

5.7 Relationship of Parsimony to Problem Size

Figure 5.8 shows the relationship between the average structural complexity of the successful programs for lawn sizes of 32, 64, and 96 both with and without automatic function definition.

As can be seen, the relationship is approximately linear for the curves with and without automatic function definition; however, the relationships are very different.

When we perform a least squares linear regression on the three-point curve without automatic function definition, we find that structural complexity S can be stated in terms of lawn size L_S by

$$S = 2.4 + 4.4L$$

The vertical intercept is small and the slope is 4.4. In other words, a small and essentially zero size is associated with a solution for a lawn of zero size and there is the rather steep 4.4 growth in the program size with the lawn size. That is, there was no economy of scale without automatic function definition. Increments in lawn size are reflected in substantial linear growth in program size.

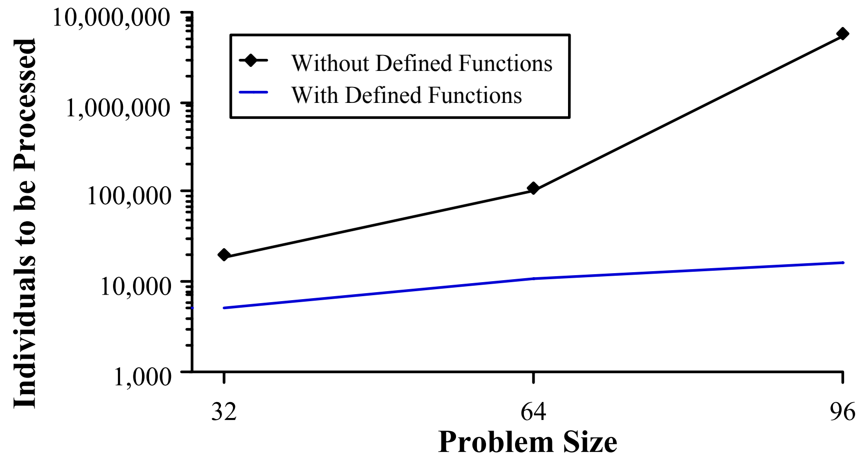

Figure 5.9
Comparison of *I(M,i,z)*, for lawn sizes of 32, 64, and 96 both with and without automatic function definition.

In contrast, when we perform the least squares linear regression on the three-point curve with automatic function definition, we find that structural complexity S can be stated in terms of lawn size L_S by

$$S = 57.85 + 0.28L_S$$

The vertical intercept has the substantial non-zero value of 57.85, but the slope is the very gentle value of 0.28. That is, there seems be a significant fixed minimum overhead associated with automatic function definition and relatively little additional cost associated with growth in the size of the problem. That is, there is a considerable economy of scale associated with automatic function definition.

5.8 Relationship of Computational Effort to Problem Size

Figure 5.9 shows the relationship between the number of fitness evaluations, *I(M,i,z)*, for lawn sizes of 32, 64, and 96 both with and without automatic function definition. Note that the vertical axis for *I(M,i,z)* uses a logarithmic scale.

As can be seen, the progression of values of *I(M,i,z)* with lawn size without automatic function definition from 19,000 to 100,000 to 4,539,000 is a steeply nonlinear pattern of

growth. This nonlinearity is even greater than it may first appear by inspection of the graph because of the logarithmic scale.

In contrast, the progression of values of *I(M,i,z)* with automatic function definition from 5,000 to 11,000 to 16,000 is a nearly linear relationship based on the problem sizes of 32, 64, and 96. In fact, when we perform the least squares linear regression on the three-point curve with automatic function definition, we find that the number of individuals I can be stated in terms of lawn size L_S by

$$I = -333 + 172 L_S$$

with a correlation R of 1.00.

5.9 Conclusions

This chapter considered a problem with substantial symmetry and regularity in its problem environment. Three differently sized versions of the problem were solved both with and without automatic function definition.

For a fixed lawn size of 64, substantially fewer fitness evaluations are required to yield a solution of the problem with 99% probability with automatic function definition than without it. Moreover, the average size of the programs that successfully solved the problem is considerably smaller with automatic function definition than without it.

When the problem size was varied upwards and downwards by 50% from 64 without automatic function definition, the average size of the programs that successfully solved the problem was almost a direct linear multiple of the problem size and there is no economy of scale without automatic function definition. However, with automatic function definition, the average size of the programs that successfully solved the problem started with a certain fixed overhead and increased only gently with problem size.

There is a steeply nonlinear pattern to the number of fitness evaluations required to yield a solution to the problem with 99% probability without automatic function definition when the problem size is varied upwards and downwards by 50% from 64. However, with automatic function definition, there is only a gentle linear growth in the number of fitness evaluations.

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.

Bibliography

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

Koza, J. R. (1992b) 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.

Koza, J. R. (1993) 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.

Koza, J. R. (1994) Genetic Programming II. Cambridge, MA: The MIT Press. In Press.

Koza, J. R., M. A. Keane and J. P. Rice (1993), 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.

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

Koza, J. R., and J. P. Rice (1994), Genetic Programming II Videotape: The Next Generation. Cambridge, MA: The MIT Press. In Press.