2 Background on Genetic Programming

This chapter provides basic background information on genetic programming.

Genetic programming is a domain-independent method that genetically breeds a population of computer programs to solve a problem. Specifically, genetic programming iteratively transforms a population of computer programs into a new generation of programs by applying analogs of naturally occurring genetic operations. The genetic operations include crossover (sexual recombination), mutation, reproduction, gene duplication, and gene deletion. Analogs of developmental processes that transform an embryo into a fully developed entity are also employed. Genetic programming is an extension of the genetic algorithm (Holland 1975) into the arena of computer programs.

This chapter describes

- the preparatory steps of genetic programming (section 2.1),
- the executional steps (flowchart) of genetic programming (section 2.2),
- advanced features of genetic programming (section 2.3),
- the main points of our previous books on genetic programming (section 2.4), and
- sources of additional information about genetic programming (section 2.5).

2.1 Preparatory Steps of Genetic Programming

Genetic programming starts from a high-level statement of the requirements of a problem and attempts to produce a computer program that solves the problem.

The human user communicates the high-level statement of the problem to the genetic programming system by performing certain well-defined preparatory steps.

The five major preparatory steps for the basic version of genetic programming are described in Genetic Programming: On the Programming of Computers by Means of Natural Selection (Koza 1992a). They require the human user to specify

- (1) the set of terminals (e.g., the independent variables of the problem, zero-argument functions, and random constants) for each branch of the to-be-evolved program,
- (2) the set of primitive functions for each branch of the to-be-evolved program,
(3) the fitness measure (for explicitly or implicitly measuring the fitness of individuals in the population),
(4) certain parameters for controlling the run, and
(5) the termination criterion and method for designating the result of the run.

Figure 1.1 shows the five major preparatory steps for the basic version of genetic programming. The preparatory steps (shown at the top of the figure) are the human-supplied input to the genetic programming system. The computer program (shown at the bottom) is the output of the genetic programming system.

The first two preparatory steps specify the ingredients that are available to create the computer programs. A run of genetic programming is a competitive search among a diverse population of programs composed of the available functions and terminals.

The identification of the function set and terminal set for a particular problem (or category of problems) is usually a straightforward process. For some problems, the function set may consist of merely the arithmetic functions of addition, subtraction, multiplication, and division as well as a conditional branching operator. The terminal set may consist of the program’s external inputs (independent variables) and numerical constants.

For many other problems, the ingredients include specialized functions and terminals. For example, if the goal is to get genetic programming to automatically program a robot to mop the entire floor of an obstacle-laden room, the human user must tell genetic programming what the robot is capable of doing. For example, the robot may be capable of executing functions such as moving, turning, and swishing the mop.

If the goal is the automatic creation of a controller, the function set may consist of integrators, differentiators, leads, lags, gains, adders, subtractors, and the like. The terminal set may consist of signals such as the reference signal and plant output.

If the goal is the automatic synthesis of an analog electrical circuit, the function set may enable genetic programming to construct circuits from components such as transistors, capacitors, and resistors. Once the human user has identified the primitive ingredients for a problem of circuit synthesis, the same function set can be used to automatically synthesize an amplifier, computational circuit, active filter, voltage reference circuit, or any other circuit composed of these ingredients.

The third preparatory step concerns the fitness measure for the problem. The fitness measure specifies what needs to be done. The fitness measure is the primary mechanism for communicating the high-level statement of the problem’s requirements to the genetic programming system. For example, if the goal is to get genetic programming to automatically synthesize an amplifier, the fitness function is the mechanism for telling genetic programming to synthesize a circuit that amplifies an incoming signal (as opposed to, say, a circuit that suppresses the low frequencies of an incoming signal or that computes the square root of the incoming signal). The first two preparatory steps define the search space whereas the fitness measure implicitly specifies the search’s desired goal.

The fourth and fifth preparatory steps are administrative. The fourth preparatory step entails specifying the control parameters for the run. The most important control parameter is the population size. In this book, we normally choose a population size that will produce a reasonably large number of generations in the amount of computer
time we are willing to devote to a problem (as opposed to analytically choosing the population size by somehow analyzing a problem’s fitness landscape). Other control parameters include the probabilities of performing the genetic operations, the maximum size for programs, and other details of the run.

The fifth preparatory step consists of specifying the termination criterion and the method of designating the result of the run. The termination criterion may include a maximum number of generations to be run as well as a problem-specific success predicate. Our practice for all problems in this book is to manually monitor and manually terminate the run when the values of fitness for numerous successive best-of-generation individuals appear to have reached a plateau. The single best-so-far individual is then harvested and designated as the result of the run.

2.2 Executional Steps of Genetic Programming

After the user has performed the preparatory steps for a problem, the run of genetic programming can be launched. Once the run is launched, a series of well-defined, problem-independent steps is executed.

Genetic programming typically starts with a population of randomly generated computer programs composed of the available programmatic ingredients (as provided by the human user in the first and second preparatory steps).

Genetic programming iteratively transforms a population of computer programs into a new generation of the population by applying analogs of naturally occurring genetic operations. These operations are applied to individual(s) selected from the population. The individuals are probabilistically selected to participate in the genetic operations based on their fitness (as measured by the fitness measure provided by the human user in the third preparatory step). The iterative transformation of the population is executed inside the main generational loop of the run of genetic programming.

The executional steps of genetic programming are as follows:

1. Randomly create an initial population (generation 0) of individual computer programs composed of the available functions and terminals.
2. Iteratively perform the following sub-steps (called a generation) on the population until the termination criterion is satisfied:
   a. Execute each program in the population and ascertain its fitness (explicitly or implicitly) using the problem’s fitness measure.
   b. Select one or two individual program(s) from the population with a probability based on fitness (with reselection allowed) to participate in the genetic operations in (c).
   c. Create new individual program(s) for the population by applying the following genetic operations with specified probabilities:
      i. **Reproduction**: Copy the selected individual program to the new population.
      ii. **Crossover**: Create new offspring program(s) for the new population by recombining randomly chosen parts from two selected programs.
      iii. **Mutation**: Create one new offspring program for the new population by randomly mutating a randomly chosen part of one selected program.
(iv) **Architecture-altering operations**: Choose an architecture-altering operation from the available repertoire of such operations and create one new offspring program for the new population by applying the chosen architecture-altering operation to one selected program.

(3) After the termination criterion is satisfied, the single best program in the population produced during the run (the best-so-far individual) is harvested and designated as the result of the run. If the run is successful, the result may be a solution (or approximate solution) to the problem.

Figure 2.1 is a flowchart of genetic programming showing the genetic operations of crossover, reproduction, and mutation as well as the architecture-altering operations. This flowchart shows a two-offspring version of the crossover operation (used in the example run in section 2.2.1). One-offspring crossover is used for all the other problems this book.

![Flowchart of genetic programming](image-url)

**Figure 2.1** Flowchart of genetic programming.
The preparatory steps specify what the user must provide in advance to the genetic programming system. Once the run is launched, the executional steps as shown in the flowchart (figure 2.1) are executed. Genetic programming is problem-independent in the sense that the flowchart specifying the basic sequence of executional steps is not modified for each new run or each new problem.

There is usually no discretionary human intervention or interaction during a run of genetic programming (although a human user may exercise judgment as to whether to terminate a run).

Genetic programming starts with an initial population of computer programs composed of functions and terminals appropriate to the problem. The individual programs in the initial population are typically generated by recursively generating a rooted point-labeled program tree composed of random choices of the primitive functions and terminals (provided by the user as part of the first and second preparatory steps). The initial individuals are usually generated subject to a pre-established maximum size (specified by the user as a minor parameter as part of the fourth preparatory step). In general, the programs in the population are of different size (number of functions and terminals) and of different shape (the particular graphical arrangement of functions and terminals in the program tree).

Each individual program in the population is executed. Then, each individual program in the population is either measured or compared in terms of how well it performs the task at hand (using the fitness measure provided in the third preparatory step). For many problems (including all problems in this book), this measurement yields a single explicit numerical value, called fitness. The fitness of a program may be measured in many different ways, including, for example, in terms of the amount of error between its output and the desired output, the amount of time (fuel, money, etc.) required to bring a system to a desired target state, the accuracy of the program in recognizing patterns or classifying objects into classes, the payoff that a game-playing program produces, or the compliance of a complex structure (such as an antenna, circuit, or controller) with user-specified design criteria. The execution of the program sometimes returns one or more explicit values. Alternatively, the execution of a program may consist only of side effects on the state of a world (e.g., a robot’s actions). Alternatively, the execution of a program may produce both return values and side effects.

The fitness measure is, for many practical problems, multiobjective in the sense that it combines two or more different elements. The different elements of the fitness measure are often in competition with one another to some degree.

For many problems, each program in the population is executed over a representative sample of different fitness cases. These fitness cases may represent different values of the program’s input(s), different initial conditions of a system, or different environments. Sometimes the fitness cases are constructed probabilistically.

The creation of the initial random population is, in effect, a blind random search of the search space of the problem. It provides a baseline for judging future search efforts. Typically, the individual programs in generation 0 all have exceedingly poor fitness. Nonetheless, some individuals in the population are (usually) more fit than others. The differences in fitness are then exploited by genetic programming. Genetic programming applies Darwinian selection and the genetic operations to create a new population of offspring programs from the current population.
The genetic operations include crossover (sexual recombination), mutation, reproduction, and the architecture-altering operations. These genetic operations are applied to individual(s) that are probabilistically selected from the population based on fitness. In this probabilistic selection process, better individuals are favored over inferior individuals. However, the best individual in the population is not necessarily selected and the worst individual in the population is not necessarily passed over.

After the genetic operations are performed on the current population, the population of offspring (i.e., the new generation) replaces the current population (i.e., the now-old generation). This iterative process of measuring fitness and performing the genetic operations is repeated over many generations.

The run of genetic programming terminates when the termination criterion (as provided by the fifth preparatory step) is satisfied. The outcome of the run is specified by the method of result designation. The best individual ever encountered during the run (i.e., the best-so-far individual) is typically designated as the result of the run.

There are numerous alternative implementations of genetic programming that vary from the preceding brief description.

2.2.1 Example of a Run of Genetic Programming

To provide concreteness, this section contains an illustrative run of genetic programming in which the goal is to automatically create a computer program whose output is equal to the values of the quadratic polynomial \(x^2 + x + 1\) in the range from \(-1\) to \(+1\). That is, the goal is to automatically create a computer program that matches certain numerical data. This process is sometimes called system identification or symbolic regression.

We begin with the five preparatory steps.

The purpose of the first two preparatory steps is to specify the ingredients of the to-be-evolved program.

Because the problem is to find a mathematical function of one independent variable, the terminal set (inputs to the to-be-evolved program) includes the independent variable, \(x\). The terminal set also includes numerical constants. That is, the terminal set, \(T\), is

\[ T = \{ x, \mathbb{R} \}, \]

where \(\mathbb{R}\) denotes constant numerical terminals in some reasonable range (say from \(-5.0\) to \(+5.0\)).

The preceding statement of the problem is somewhat flexible in that it does not specify what functions may be employed in the to-be-evolved program. One possible choice for the function set consists of the four ordinary arithmetic functions of addition, subtraction, multiplication, and protected division. This choice is reasonable because mathematical expressions typically include these functions. Thus, the function set, \(F\), for this problem is

\[ F = \{ +, -, *, \% \}. \]
The two-argument $+, -, \ast, \%$ functions add, subtract, multiply, and divide, respectively. The protected division function $\%$ returns a value of 1 when division by 0 is attempted (including 0 divided by 0), but otherwise returns the quotient of its two arguments.

Each individual in the population is a composition of functions from the specified function set and terminals from the specified terminal set.

The third preparatory step involves constructing the fitness measure. The purpose of the fitness measure is to specify what the human wants. The high-level goal of this problem is to find a program whose output is equal to the values of the quadratic polynomial $x^2 + x + 1$. Therefore, the fitness assigned to a particular individual in the population for this problem must reflect how closely the output of an individual program comes to the target polynomial $x^2 + x + 1$. The fitness measure could be defined as the value of the integral (taken over values of the independent variable $x$ between $-1.0$ and $+1.0$) of the absolute value of the differences (errors) between the value of the individual mathematical expression and the target quadratic polynomial $x^2 + x + 1$. A smaller value of fitness (error) is better. A fitness (error) of zero would indicate a perfect fit.

For most problems of symbolic regression or system identification, it is not practical or possible to analytically compute the value of the integral of the absolute error. Thus, in practice, the integral is numerically approximated using dozens or hundreds of different values of the independent variable $x$ in the range between $-1.0$ and $+1.0$.

The population size in this small illustrative example will be just four. In actual practice, the population size for a run of genetic programming consists of thousands or millions of individuals. In actual practice, the crossover operation is commonly performed on about 90% of the individuals in the population; the reproduction operation is performed on about 8% of the population; the mutation operation is performed on about 1% of the population; and the architecture-altering operations are performed on perhaps 1% of the population. Because this illustrative example involves an abnormally small population of only four individuals, the crossover operation will be performed on two individuals and the mutation and reproduction operations will each be performed on one individual. For simplicity, the architecture-altering operations are not used for this problem.

A reasonable termination criterion for this problem is that the run will continue from generation to generation until the fitness of some individual gets below 0.01. In this contrived example, the run will (atypically) yield an algebraically perfect solution (for which the fitness measure attains the ideal value of zero) after merely one generation.

Now that we have performed the five preparatory steps, the run of genetic programming can be launched. That is, the executional steps shown in the flowchart of figure 2.1 are now performed.

Genetic programming starts by randomly creating a population of four individual computer programs. The four programs are shown in figure 2.2 in the form of trees.

The first randomly constructed program tree (figure 2.2a) is equivalent to the mathematical expression $x + 1$. A program tree is executed in a depth-first way, from left to right, in the style of the LISP programming language. Specifically, the addition function $(\ast)$ is executed with the variable $x$ and the constant value 1 as its two arguments. Then, the two-argument subtraction function $(\%)$ is executed. Its first argument is the value returned by the just-executed addition function. Its second argument is the constant value 0. The overall result of executing the entire program tree is thus $x + 1$. 

Background on Genetic Programming

35
The first program (figure 2.2a) was constructed by first choosing the subtraction function for the root (top point) of the program tree. The random construction process continued in a depth-first fashion (from left to right) and chose the addition function to be the first argument of the subtraction function. The random construction process then chose the terminal $x$ to be the first argument of the addition function (thereby terminating the growth of this path in the program tree). The random construction process then chose the constant terminal 1 as the second argument of the addition function (thereby terminating the growth along this path). Finally, the random construction process chose the constant terminal 0 as the second argument of the subtraction function (thereby terminating the entire construction process).

The second program (figure 2.2b) adds the constant terminal 1 to the result of multiplying $x$ by $x$ and is equivalent to $x^2 + 1$. The third program (figure 2.2c) adds the constant terminal 2 to the constant terminal 0 and is equivalent to the constant value 2. The fourth program (figure 2.2d) is equivalent to $x$.

$$\begin{align*}
(a) & \quad (b) & \quad (c) & \quad (d) \\
& \quad + & \quad + & \quad + \\
& \quad 0 & \quad 1 & \quad 2 & \quad x \\
x & \quad 1 & \quad x & \quad x & \quad - \\
& \quad x+1 & \quad x^2+1 & \quad 2 & \quad x \quad -1 & \quad -2
\end{align*}$$

Figure 2.2 Initial population of four randomly created individuals of generation 0.

$$\begin{align*}
(a) & \quad (b) & \quad (c) & \quad (d) \\
& \quad + & \quad % & \quad - \\
& \quad 0 & \quad 0 & \quad 0 & \quad x \\
x & \quad 1 & \quad x & \quad x & \quad x \quad 1 \\
& \quad x+1 & \quad 1 & \quad x & \quad x^2 + x + 1
\end{align*}$$

Figure 2.4 Population of generation 1 (after one reproduction, one mutation, and one two-offspring crossover operation).
Randomly created computer programs will, of course, typically be very poor at solving the problem at hand. However, even in a population of randomly created programs, some programs are better than others. The four random individuals from generation 0 in figure 2.2 produce outputs that deviate from the output produced by the target quadratic function \( x^2 + x + 1 \) by different amounts. In this particular problem, fitness can be graphically illustrated as the area between two curves. That is, fitness is equal to the area between the parabola \( x^2 + x + 1 \) and the curve representing the candidate individual. Figure 2.3 shows (as shaded areas) the integral of the absolute value of the errors between each of the four individuals in figure 2.2 and the target quadratic function \( x^2 + x + 1 \). The integral of absolute error for the straight line \( x + 1 \) (the first individual) is 0.67 (figure 2.3a). The integral of absolute error for the parabola \( x^2 + 1 \) (the second individual) is 1.0 (figure 2.3b). The integral of the absolute errors for the remaining two individuals are 1.67 (figure 2.3c) and 2.67 (figure 2.3d), respectively.

As can be seen in figure 2.3, the straight line \( x + 1 \) (figure 2.3a) is closer to the parabola \( x^2 + x + 1 \) in the range from \(-1\) to \(+1\) than any of its three cohorts in the population. This straight line is, of course, not equivalent to the parabola \( x^2 + x + 1 \). This best-of-generation individual from generation 0 is not even a quadratic function. It is merely the best candidate that happened to emerge from the blind random search of generation 0. In the valley of the blind, the one-eyed man is king.

After the fitness of each individual in the population is ascertained, genetic programming then probabilistically selects relatively more fit programs from the population. The genetic operations are applied to the selected individuals to create offspring programs. The most commonly employed methods for selecting individuals to participate in the genetic operations are tournament selection (used throughout this book) and fitness-proportionate selection. In both methods (described in Koza 1992a), the emphasis is on selecting relatively fit individuals. An important feature common to both methods is that the selection is not greedy. Individuals that are known to be inferior will be selected to a certain degree. The best individual in the population is not guaranteed to be selected. Moreover, the worst individual in the population will not necessarily be excluded. Anything can happen and nothing is guaranteed.

We first perform the reproduction operation. Because the first individual (figure 2.2a) is the most fit individual in the population, it is very likely to be selected to participate in a genetic operation. Let’s suppose that this particular individual is, in fact, selected for reproduction. If so, it is copied, without alteration, into the next generation (generation 1). It is shown in figure 2.4a as part of the population of the new generation.

We next perform the mutation operation. Because selection is probabilistic, it is possible that the third best individual in the population (figure 2.2c) is selected. One of the three points of this individual is then randomly picked as the site for the mutation. In this example, the constant terminal 2 is picked as the mutation site. This program is then randomly mutated by deleting the entire subtree rooted at the picked point (in this case, just the constant terminal 2) and inserting a subtree that is randomly grown in the same way that the individuals of the initial random population were originally created. In this particular instance, the randomly grown subtree (shown in figure 2.4b) computes the quotient of \( x \) and \( x \) using the protected division
operation %. This particular mutation changes the original individual from one having a constant value of 2 into one having a constant value of 1. This particular mutation improves fitness from 1.67 to 1.00.

Finally, we perform the crossover operation. Because the first and second individuals in generation 0 are both relatively fit, they are likely to be selected to participate in crossover. In fact, because of its high fitness, the first individual ends up being selected twice to participate in a genetic operation. In contrast, the unfit fourth individual is not be selected at all. The reselection of relatively more fit individuals and the exclusion and extinction of unfit individuals is a characteristic feature of Darwinian selection. The first and second programs are mated sexually to produce two offspring (using the two-offspring version of the crossover operation). One point of the first parent (figure 2.2a), namely the + function, is randomly picked as the crossover point for the first parent. One point of the second parent (figure 2.2b), namely its leftmost terminal \( x \), is randomly picked as the crossover point for the second parent. The crossover operation is then performed on the two parents. The two offspring are shown in figures 2.4c and 2.4d. One of the offspring (figure 2.4c) is equivalent to \( x \) and is not noteworthy. However, the other offspring (figure 2.4d) is equivalent to \( x^2 + x + 1 \) and has a fitness (integral of absolute errors) of zero. Because the fitness of this individual is below 0.01, the termination criterion for the run is satisfied and the run is automatically terminated. This best-so-far individual (figure 2.4d) is designated as the result of the run. This individual is an algebraically correct solution to the problem.

Note that the best-of-run individual (figure 2.4d) incorporates a good trait (the quadratic term \( x^2 \)) from the second parent (figure 2.2b) with two other good traits (the linear term \( x \) and constant term of 1) from the first parent (figure 2.2a). The crossover operation produced a solution to this problem by recombining good traits from these two relatively fit parents into a superior (indeed, perfect) offspring.

In summary, genetic programming has, in this example, automatically created a computer program whose output is equal to the values of the quadratic polynomial \( x^2 + x + 1 \) in the range from \(-1\) to \(1\).

Additional details of the operation of basic genetic programming are found in *Genetic Programming: On the Programming of Computers by Means of Natural Selection* (Koza 1992a).

### 2.3 Advanced Features of Genetic Programming

Various advanced features of genetic programming are not covered by the foregoing illustrative problem and the foregoing discussion of the preparatory and executional steps of genetic programming.

#### 2.3.1 Constrained Syntactic Structures

For certain simple problems (such as the illustrative problem in section 2.2.1), the search space for a run of genetic programming consists of the unrestricted set of possible compositions of the problem’s functions and terminals.
However, for many problems (including every problem in this book), a constrained syntactic structure imposes restrictions on how the functions and terminals may be combined.

Consider, for example, a function that instructs a robot to turn by a certain angle. In a typical implementation of this hypothetical function, the function’s first argument may be required to return a numerical value (representing the desired turning angle) and its second argument may be required to be a follow-up command (e.g., move, turn, stop). In other words, the functions and terminals permitted in the two argument subtrees for this particular function are restricted. These restrictions are implemented by means of syntactic rules of construction.

A constrained syntactic structure (sometimes called strong typing) is a grammar that specifies the functions or terminals that are permitted to appear as a specified argument of a specified function in the program tree.

When a constrained syntactic structure is used, there are typically multiple function sets and multiple terminal sets. The rules of construction specify where the different function sets or terminal sets may be used.

When a constrained syntactic structure is used, all the individuals in the initial random population (generation 0) are created so as to comply with the constrained syntactic structure. All genetic operations (i.e., crossover, mutation, reproduction, and the architecture-altering operations) that are performed during the run are designed to produce offspring that comply with the requirements of the constrained syntactic structure. Thus, all individuals (including, in particular, the best-of-run individual) that are produced during the run of genetic programming will necessarily comply with the requirements of the constrained syntactic structure.

### 2.3.2 Automatically Defined Functions

Human computer programmers organize sequences of reusable steps into subroutines. They then repeatedly invoke the subroutines—typically with different instantiations of the subroutine’s dummy variables (formal parameters). Reuse eliminates the need to “reinvent the wheel” on each occasion when a particular sequence of steps may be needed. Reuse makes it possible to exploit a problem’s modularities, symmetries, and regularities (and thereby potentially accelerate the problem-solving process).

Programmers commonly organize their subroutines into hierarchies.

The automatically defined function (ADF) is one of the mechanisms by which genetic programming implements the parameterized reuse and hierarchical invocation of evolved code. Each automatically defined function resides in a separate function-defining branch within the overall multi-part computer program. When automatically defined functions are being used, a program consists of one (or more) function-defining branches (i.e., automatically defined functions) as well as one or more main result-producing branches. An automatically defined function may possess zero, one, or more dummy variables (formal parameters). The body of an automatically defined function contains its work-performing steps. Each automatically defined function belongs to a particular program in the population. An automatically defined function may be called by the program’s main result-producing branch, another automatically defined function, or another type of branch (such as those described in section 2.3.3).
Recursion may be allowed. Typically, the automatically defined functions are invoked with different instantiations of their dummy variables.

The work-performing steps of the program’s main result-producing branch and the work-performing steps of each automatically defined function are automatically and simultaneously created during the run of genetic programming.

The program’s main result-producing branch and its automatically defined functions typically have different function and terminal sets. A constrained syntactic structure (section 2.3.1) is used to implement automatically defined functions.

Automatically defined functions are the focus of Genetic Programming II: Automatic Discovery of Reusable Programs (Koza 1994a) and the videotape Genetic Programming II Videotape: The Next Generation (Koza 1994b). See also Koza 1990a, 1992a, 1992c; Koza and Rice 1991, 1992b, 1994a; and Koza, Bennett, Andre, and Keane 1999a.

### 2.3.3 Automatically Defined Iterations, Automatically Defined Loops, Automatically Defined Recursions, and Automatically Defined Stores

Automatically defined iterations (ADIs), automatically defined loops (ADLs), and automatically defined recursions (ADRs) provide means (in addition to automatically defined functions) to reuse code.

Automatically defined stores (ADSs) provide means to reuse the result of executing code.

Automatically defined iterations, automatically defined loops, automatically defined recursions, and automatically defined stores are described in Genetic Programming III: Darwinian Invention and Problem Solving (Koza, Bennett, Andre, and Keane 1999a).

### 2.3.4 Program Architecture and Architecture-Altering Operations

The architecture of a program consists of

- the total number of branches,
- the type of each branch (e.g., result-producing branch, automatically defined function, automatically defined iteration, automatically defined loop, automatically defined recursion, or automatically defined store),
- the number of arguments (if any) possessed by each branch, and
- if there is more than one branch, the nature of the hierarchical references (if any) allowed among the branches.

There are three ways by which genetic programming can arrive at the architecture of the to-be-evolved computer program:

- The human user may prespecify the architecture of the overall program (i.e., perform an additional architecture-defining preparatory step). That is, the number of preparatory steps is increased from the five itemized in section 2.1 to six.
- The run may employ evolutionary selection of the architecture (as described in Genetic Programming II), thereby enabling the architecture of the overall program
to emerge from a competitive process during the run of genetic programming. When this approach is used, the number of preparatory steps remains at the five itemized in section 2.1.

- The run may employ the architecture-altering operations (Koza 1994c, 1995a, 1995b, 1995c; Koza, Andre, and Tackett 1994, 1998; Koza, Bennett, Andre, and Keane 1999a), thereby enabling genetic programming to automatically create the architecture of the overall program dynamically during the run. When this approach is used, the number of preparatory steps remains at the five itemized in section 2.1.

2.3.5 Genetic Programming Problem Solver (GPPS)


If GPPS is being used, the user is relieved of performing the first and second preparatory steps (concerning the choice of the terminal set and the function set). The function set for GPPS consists of the four basic arithmetic functions (addition, subtraction, multiplication, and division) and a conditional operator (i.e., functions found in virtually every general-purpose digital computer that has ever been built). The terminal set for GPPS consists of numerical constants and a set of input terminals that are presented in the form of a vector.

By employing this generic function set and terminal set, GPPS reduces the number of preparatory steps from five to three.

GPPS relies on the architecture-altering operations to dynamically create, duplicate, and delete subroutines and loops during the run of genetic programming. Additionally, in version 2.0 of GPPS, the architecture-altering operations are used to dynamically create, duplicate, and delete recursions and internal storage. Because the architecture of the evolving program is automatically determined during the run, GPPS eliminates the need for the user to specify in advance whether to employ subroutines, loops, recursions, and internal storage in solving a given problem. It similarly eliminates the need for the user to specify the number of arguments possessed by each subroutine. And, GPPS eliminates the need for the user to specify the hierarchical arrangement of the invocations of the subroutines, loops, and recursions. That is, the use of GPPS relieves the user of performing the preparatory step of specifying the program’s architecture. The results produced by GPPS include one human-competitive result (section 23.6 of *Genetic Programming III*). However, most of the problems solved using GPPS have, as of this writing, been toy problems comparable in difficulty to those found in the 1992 book *Genetic Programming* (Koza 1992a). See also Koza, Bennett, Andre, and Keane 1999c.

2.3.6 Developmental Genetic Programming

Developmental genetic programming is used for problems of synthesizing analog electrical circuits, as described in chapters 4, 5, 10, 11, 14, and 15 of this book and in part 5 of *Genetic Programming III*. When developmental genetic programming is used, a complex structure (such as an electrical circuit) is created from a simple initial structure (the embryo). If the user desires to specify the embryo, the number of
preparatory steps is increased from five to six. On the other hand, the user can be relieved of specifying a particular embryo for a problem by using a generic floating embryo (section 4.7.1.1). Early work on development includes Kitano’s (1990) use of genetic algorithms to evolve neural networks, Gruau’s work on cellular encoding (developmental genetic programming) to evolve neural networks (1992a, 1992b), work by Spector and Stoffel on ontogenetic programming (1996a, 1996b), and work involving the evolution of Lindenmayer rules for creating structures (Koza 1993).

2.3.7 Computer Code for Implementing Genetic Programming

Genetic programming has been implemented in numerous programming languages. Most present-day versions of genetic programming are written in C, C++, or Java. LISP code for implementing genetic programming is available in Genetic Programming (Koza 1992a). Web sites such as www.genetic-programming.org contain links to computer code in various other programming languages.

2.4 Main Points of Four Books on Genetic Programming

Table 2.1 shows the main points of our four books on genetic programming. The two main points of the first book and the eight main points of the second book have been paraphrased for purposes of this table (but are quoted in full later in this section). As can be seen, the main points of this book fit into a progression of results starting with the 1992 book.

<table>
<thead>
<tr>
<th>Book</th>
<th>Main points</th>
</tr>
</thead>
</table>
| 1992 | ● Virtually all problems in artificial intelligence, machine learning, adaptive systems, and automated learning can be recast as a search for a computer program.  
      | ● Genetic programming provides a way to successfully conduct the search for a computer program in the space of computer programs. |
| 1994 | ● Scalability is essential for solving non-trivial problems in artificial intelligence, machine learning, adaptive systems, and automated learning.  
      | ● Scalability can be achieved by reuse.  
      | ● Genetic programming provides a way to automatically discover and reuse subprograms in the course of automatically creating computer programs to solve problems. |
| 1999 | ● Genetic programming possesses the attributes that can reasonably be expected of a system for automatically creating computer programs. |
| 2003 | ● Genetic programming now routinely delivers high-return human-competitive machine intelligence.  
      | ● Genetic programming is an automated invention machine.  
      | ● Genetic programming can automatically create a general solution to a problem in the form of a parameterized topology.  
      | ● Genetic programming has delivered a progression of qualitatively more substantial results in synchrony with five approximately order-of-magnitude increases in the expenditure of computer time. |
The 1992 book *Genetic Programming: On the Programming of Computers by Means of Natural Selection* (Koza 1992a) and its accompanying videotape *Genetic Programming: The Movie* (Koza and Rice 1992a) established the principle that genetic programming is capable of automatically creating a computer program that solves a problem. The two main points of this book are:

- A wide variety of seemingly different problems from many different fields can be recast as requiring the discovery of a computer program that produces some desired output when presented with particular inputs. That is, many seemingly different problems can be reformulated as problems of program induction.
- The recently developed genetic programming paradigm described in this book provides a way to do program induction. That is, genetic programming can search the space of possible computer programs for an individual computer program that is highly fit in solving (or approximately solving) the problem at hand. The computer program (i.e., structure) that emerges from the genetic programming paradigm is a consequence of fitness. That is, fitness begets the needed program structure.

These points were demonstrated on a wide range of “toy” (proof of principle) problems. Most of these problems were benchmark problems taken from the literature of the 1980s and early 1990s from the fields of artificial intelligence, machine learning, neural networks, decision trees, and reinforcement learning. The 1992 book made the fundamental point that these seemingly different problems could all be recast as a search for a computer program. Of course, virtually all practitioners of artificial intelligence and machine learning subscribe to the Church-Turing thesis (and therefore believe that the solutions to such problems can be represented as computer programs). So, in one sense, our first book’s first main point (about recasting problems as a search for a computer program) was platitudinous. Yet, in a practical sense, this point was radical. Indeed, as far as we know, genetic programming was (at the time of its invention) unique among techniques of artificial intelligence, machine learning, adaptive systems, and automated learning in that it searches a space of ordinary computer programs. Since then, work on automatic program synthesis has included Olsson’s ADATE system (Olsson 1994a, 1994b).

In any event, once the toy problems from various fields of artificial intelligence and machine learning were recast as a search for a computer program, the book *Genetic Programming* demonstrated that they could all be solved with a single uniform method, namely genetic programming. The problems solved in *Genetic Programming* include problems of symbolic regression (system identification, empirical discovery, modeling, forecasting, data mining), classification, control, optimization, equation solving, game playing, induction, problems exhibiting emergent behavior, problems involving co-evolution, cellular automata programming, randomizer construction, image compression, symbolic integration and differentiation, inverse problems, decision tree induction, and many others.

The 1994 book *Genetic Programming II: Automatic Discovery of Reusable Programs* (Koza 1994a) and its accompanying videotape *Genetic Programming II Videotape: The Next Generation* (Koza 1994b) made the key point that the reuse of code is a critical ingredient to scalable automatic programming. The book discusses scalability
in terms of the rate (e.g., linearly, exponentially) at which the computational effort required to yield a solution to differently sized instances of a particular problem (e.g., \(n^{th}\)-order parity problem, lawnmower problem for an \(n \times m\) lawn) changes as a function of problem size. The book demonstrated that one way of achieving scalability is by reusing code by means of subroutines (automatically defined functions) and iterations (automatically defined iterations). The book’s eight main points are:

- Automatically defined functions enable genetic programming to solve a variety of problems in a way that can be interpreted as a decomposition of a problem into subproblems, a solving of the subproblems, and an assembly of the solutions to the subproblems into a solution to the overall problem (or that can alternatively be interpreted as a search for regularities in the problem environment, a change in representation, and a solving of a higher-level problem).
- Automatically defined functions discover and exploit the regularities, symmetries, homogeneities, similarities, patterns, and modularities of the problem environment in ways that are very different from the style employed by human programmers.
- For a variety of problems, genetic programming requires less computational effort (fewer fitness evaluations to yield a solution with a satisfactorily high probability) with automatically defined functions than without them, provided the difficulty of the problem is above a certain relatively low break-even point.
- For a variety of problems, genetic programming usually yields solutions with smaller overall size (lower average structural complexity) with automatically defined functions than without them, provided the difficulty of the problem is above a certain break-even point.
- For the three problems in *Genetic Programming II* for which a progression of several scaled-up versions is studied, the average size of the solutions produced by genetic programming increases as a function of problem size at a lower rate with automatically defined functions than without them.
- For the three problems in *Genetic Programming II* for which a progression of several scaled-up versions is studied, the number of fitness evaluations required by genetic programming to yield a solution (with a specified high probability) increases as a function of problem size at a lower rate with automatically defined functions than without them.
- For the three problems in *Genetic Programming II* for which a progression of several scaled-up versions is studied, the improvement in computational effort and average structural complexity conferred by automatically defined functions increases as the problem size is scaled up.
- Genetic programming is capable of simultaneously solving a problem and selecting the architecture of the overall program (consisting of the number of automatically defined functions and the number of their arguments).

In addition, *Genetic Programming II* demonstrated that it is possible to

- automatically create multibranch programs containing an iteration-performing branch as well as a main program and subroutines (e.g., the transmembrane identification problem and the omega loop problem),
automatically create multibranch programs containing multiple iteration-performing branches and iteration-terminating branches (e.g., the look-ahead version of the transmembrane problem), and

- automatically determine the architecture for a multibranch program in an architecturally diverse population by means of evolutionary selection.

Genetic Programming II contains solutions to problems from the fields of symbolic regression, control, pattern recognition, classification, computational molecular biology, and discovery of the impulse response function for an electrical circuit.

The 1999 book Genetic Programming III: Darwinian Invention and Problem Solving (Koza, Bennett, Andre, and Keane 1999a) and its accompanying videotape Genetic Programming III Videotape: Human-Competitive Machine Intelligence (Koza, Bennett, Andre, Keane, and Brave 1999) identified 16 attributes (table 2.2 of this book) that can reasonably be expected of a system for automatically creating computer programs. Genetic Programming III contains solutions to problems from the fields of system identification, time-optimal control, classification, synthesis of cellular automata rules, synthesis of minimal sorting networks, multi-agent programming, and synthesis of both the topology and sizing for analog electrical circuits. Genetic Programming III made the point that genetic programming unconditionally possesses the first 13 of these 16 attributes and that genetic programming also possesses the remaining three attributes (namely wide applicability, scalability, and human-competitiveness) to a substantial degree.

Genetic Programming III presented 14 specific instances where genetic programming automatically created a computer program that is competitive with a human-produced result. The 14 instances include two classification problems from the field of computational molecular biology, a long-standing problem involving cellular automata, a problem of synthesizing the design of a minimal sorting network, and 10 problems of synthesizing the design of both the topology and sizing of analog electrical circuits. Ten of the 14 human-competitive results in Genetic Programming III involve previously patented inventions.

2.5 Sources of Additional Information about Genetic Programming

Sources of information about genetic programming include

- Genetic Programming: On the Programming of Computers by Means of Natural Selection (Koza 1992a) and the accompanying videotape Genetic Programming: The Movie (Koza and Rice 1992a);
- Genetic Programming II: Automatic Discovery of Reusable Programs (Koza 1994a) and the accompanying videotape Genetic Programming II Videotape: The Next Generation (Koza 1994b);
- Genetic Programming III: Darwinian Invention and Problem Solving (Koza, Bennett, Andre, and Keane 1999a) and the accompanying videotape Genetic Programming III Videotape: Human-Competitive Machine Intelligence (Koza, Bennett, Andre, Keane, and Brave 1999);
Table 2.2  Sixteen attributes of a system for automatically creating computer programs

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Starts with “What needs to be done” It starts from a high-level statement specifying the requirements of the problem.</td>
</tr>
<tr>
<td>2</td>
<td>Tells us “How to do it” It produces a result in the form of a sequence of steps that satisfactorily solves the problem.</td>
</tr>
<tr>
<td>3</td>
<td>Produces a computer program It produces an entity that can run on a computer.</td>
</tr>
<tr>
<td>4</td>
<td>Automatic determination of program size It has the ability to automatically determine the number of steps that must be performed and thus does not require the user to prespecify the exact size of the solution.</td>
</tr>
<tr>
<td>5</td>
<td>Code reuse It has the ability to automatically organize useful groups of steps so that they can be reused.</td>
</tr>
<tr>
<td>6</td>
<td>Parameterized reuse It has the ability to reuse groups of steps with different instantiations of values (formal parameters or dummy variables).</td>
</tr>
<tr>
<td>7</td>
<td>Internal storage It has the ability to use internal storage in the form of single variables, vectors, matrices, arrays, stacks, queues, lists, relational memory, and other data structures.</td>
</tr>
<tr>
<td>8</td>
<td>Iterations, loops, and recursions It has the ability to implement iterations, loops, and recursions.</td>
</tr>
<tr>
<td>9</td>
<td>Self-organization of hierarchies It has the ability to automatically organize groups of steps into a hierarchy.</td>
</tr>
<tr>
<td>10</td>
<td>Automatic determination of program architecture It has the ability to automatically determine whether to employ subroutines, iterations, loops, recursions, and internal storage, and to automatically determine the number of arguments possessed by each subroutine, iteration, loop, and recursion.</td>
</tr>
<tr>
<td>11</td>
<td>Wide range of programming constructs It has the ability to implement analogs of the programming constructs that human computer programmers find useful, including macros, libraries, typing, pointers, conditional operations, logical functions, integer functions, floating-point functions, complex-valued functions, multiple inputs, multiple outputs, and machine code instructions.</td>
</tr>
<tr>
<td>12</td>
<td>Well-defined It operates in a well-defined way. It unmistakably distinguishes between what the user must provide and what the system delivers</td>
</tr>
<tr>
<td>13</td>
<td>Problem-independent It is problem-independent in the sense that the user does not have to modify the system’s executable steps for each new problem.</td>
</tr>
<tr>
<td>14</td>
<td>Wide applicability It produces a satisfactory solution to a wide variety of problems from many different fields.</td>
</tr>
<tr>
<td>15</td>
<td>Scalability It scales well to larger versions of the same problem.</td>
</tr>
<tr>
<td>16</td>
<td>Competitive with human-produced results It produces results that are competitive with those produced by human programmers, engineers, mathematicians, and designers.</td>
</tr>
</tbody>
</table>
Background on Genetic Programming

- Genetic Programming—An Introduction (Banzhaf, Nordin, Keller, and Francone 1998);
- Principia Evolvica: Simulierte Evolution mit Mathematica (Jacob 1997, in German) and Illustrating Evolutionary Computation with Mathematica (Jacob 2001);
- Genetic Programming (Iba 1996, in Japanese);
- Evolutionary Program Induction of Binary Machine Code and Its Application (Nordin 1997);
- Foundations of Genetic Programming (Langdon and Poli 2002);
- Emergence, Evolution, Intelligence: Hydroinformatics (Babovic 1996);
- Theory of Evolutionary Algorithms and Application to System Synthesis (Blickle 1997);
- edited collections of papers such as the three Advances in Genetic Programming books from the MIT Press (Kinnear 1994; Angeline and Kinnear 1996; Spector, Langdon, O’Reilly, and Angeline 1999);
- the proceedings of the Genetic Programming Conferences held between 1996 and 1998 (Koza, Goldberg, Fogel, and Riolo 1996; Koza, Deb, Dorigo, Fogel, Garzon, Iba, and Riolo 1997; Koza, Banzhaf, Chellapilla, Deb, Dorigo, Fogel, Garzon, Goldberg, Iba, and Riolo 1998);
- the proceedings of the annual Genetic and Evolutionary Computation Conference (GECCO) (combining the formerly annual Genetic Programming Conference and the formerly biannual International Conference on Genetic Algorithms) operated by the International Society for Genetic and Evolutionary Computation (ISGEC) and held starting in 1999 (Banzhaf, Daida, Eiben, Garzon, Honavar, Jakiela, and Smith 1999; Whitley, Goldberg, Cantu-Paz, Spector, Parmee, and Beyer 2000; Spector Goodman, Wu, Langdon, Voigt, Gen, Sen, Dorigo, Pezeshk, Garzon, and Burke 2001; Langdon, Cantu-Paz, Mathias, Roy, Davis, Poli, Balakrishnan, Honavar, Rudolph, Wegener, Bull, Potter, Schultz, Miller, Burke, and Jonoska 2002);
- the proceedings of the annual Euro-GP conferences held starting in 1998 (Banzhaf, Poli, Schoenauer, and Fogarty 1998; Poli, Nordin, Langdon, and Fogarty 1999; Poli, Banzhaf, Langdon, Miller, Nordin, and Fogarty 2000; Miller, Tomassini, Lanzi, Ryan, Tettamanzi, and Langdon 2001; Foster, Lutton, Miller, Ryan, and Tettamanzi 2002);
- the proceedings of the Workshop of Genetic Programming Theory and Practice organized by the Center for Study of Complex Systems of the University of Michigan (Riolo and Worzel 2003);
- the Genetic Programming and Evolvable Machines journal (from Kluwer Academic Publishers) started in April 2000;
● web sites such as www.genetic-programming.org and www.genetic-programming.com;

● early papers on genetic programming, such as the Stanford University Computer Science Department technical report *Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems* (Koza 1990a), the paper “Hierarchical Genetic Algorithms Operating on Populations of Computer Programs,” presented at the 11th International Joint Conference on Artificial Intelligence in Detroit (Koza 1989), and Koza 1988;

● an annotated bibliography of the first 100 papers on genetic programming (other than those of which John Koza was the author or co-author) in appendix F of *Genetic Programming II: Automatic Discovery of Reusable Programs* (Koza 1994a); and

● William Langdon’s bibliography on genetic programming at http://www.cs.bham.ac.uk/~wbl/biblio/ or http://liinwww.ira.uka.de/bibliography/Ai/genetic.programming.html. This bibliography is the most extensive in the field and contains over 3,034 papers (as of January 2003) by over 880 authors. It provides on-line access to many of the papers.
Genetic Programming IV
Routine Human-Competitive Machine Intelligence
Koza, J.R.; Keane, M.A.; Streeter, M.J.; Mydlowec, W.; Yu, J.; Lanza, G.
2003, XXXIV, 590 p.,