Chapter 5

GENETIC PROGRAMMING

John R. Koza
Stanford University
Stanford, CA, USA

Riccardo Poli
Department of Computer Science
University of Essex, UK

5.1 INTRODUCTION

The goal of getting computers to automatically solve problems is central to artificial intelligence, machine learning, and the broad area encompassed by what Turing called "machine intelligence" (Turing, 1948, 1950). In his talk entitled *AI: Where It Has Been and Where It Is Going*, machine learning pioneer Arthur Samuel stated the main goal of the fields of machine learning and artificial intelligence:

> [T]he aim [is] ... to get machines to exhibit behavior, which if done by humans, would be assumed to involve the use of intelligence.
>
> (Samuel, 1983)

Genetic programming is a systematic method for getting computers to automatically solve a problem starting from a high-level statement of what needs to be done. 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. This process is illustrated in Figure 5.1.

The genetic operations include crossover (sexual recombination), mutation, reproduction, gene duplication, and gene deletion. Analogs of developmental processes are sometimes used to transform an embryo into a fully developed structure. Genetic programming is an extension of the genetic algorithm (Holland, 1975), see Chapter 4, in which the *structures* in the population are not

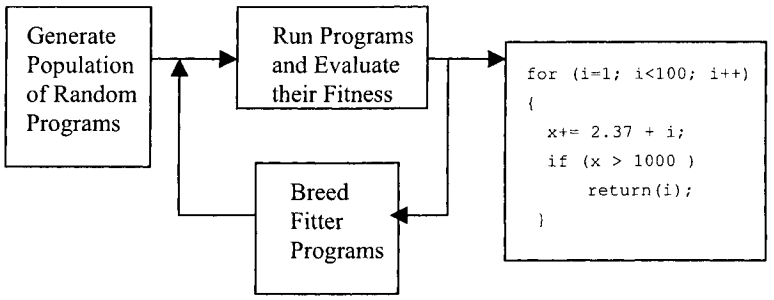

Figure 5.1. Main loop of genetic programming.

fixed-length character strings that encode candidate solutions to a problem, but *programs* that, when executed, *are* the candidate solutions to the problem.

Programs are expressed in genetic programming as *syntax trees* rather than as lines of code. For example, the simple expression

$$\mathtt{max}(\mathtt{x} * \mathtt{x}, \mathtt{x} + 3 * \mathtt{y})$$

is represented as shown in Figure 5.2. The tree includes *nodes* (which we will also call *points*) and *links*. The nodes indicate the instructions to execute. The links indicate the arguments for each instruction. In the following the internal nodes in a tree will be called *functions*, while the tree's leaves will be called *terminals*.

In more advanced forms of genetic programming, programs can be composed of multiple components (e.g. subroutines). In this case the representation used in genetic programming is a set of trees (one for each component) grouped together under a special node called *root*, as illustrated in Figure 5.3. We will call these (sub)trees *branches*. The number and type of the branches in a program, together with certain other features of the structure of the branches, form the *architecture* of the program.

Genetic programming trees and their corresponding expressions can equivalently be represented in *prefix notation* (e.g. as Lisp S-expressions). In prefix notation, functions always precede their arguments. For example, $\mathtt{max}(\mathtt{x} * \mathtt{x}, \mathtt{x} + 3 * \mathtt{y})$ becomes

$$(\mathtt{max}(* \mathtt{x}\, \mathtt{x})(+ \mathtt{x}\, (* \mathtt{3}\, \mathtt{y})))$$

In this notation, it is easy to see the correspondence between expressions and their syntax trees. Simple recursive procedures can convert prefix-notation expressions into infix-notation expressions and vice versa. Therefore, in the following, we will use trees and their corresponding prefix-notation expressions interchangeably.

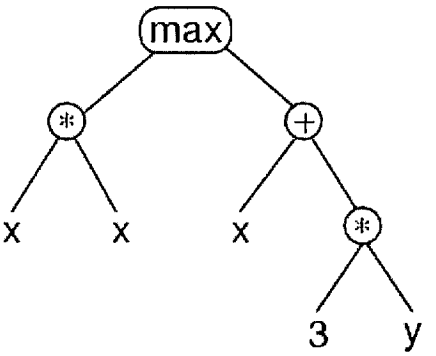

Figure 5.2. Basic tree-like program representation used in genetic programming.

5.2 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 algorithm by performing certain well-defined preparatory steps.

The five major preparatory steps for the basic version of genetic programming 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.

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

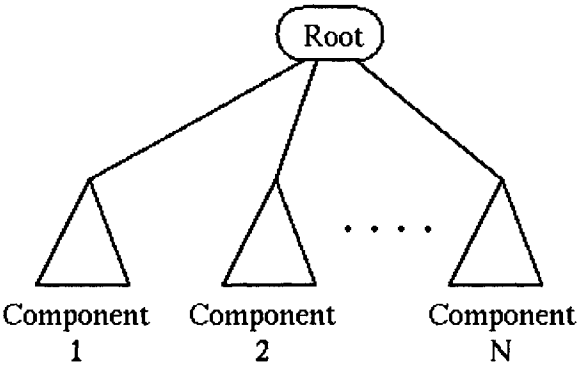

Figure 5.3. Multi-tree program representation.

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 and 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 sup-

presses 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. 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. The single best-so-far individual is then harvested and designated as the result of the run.

5.3 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).

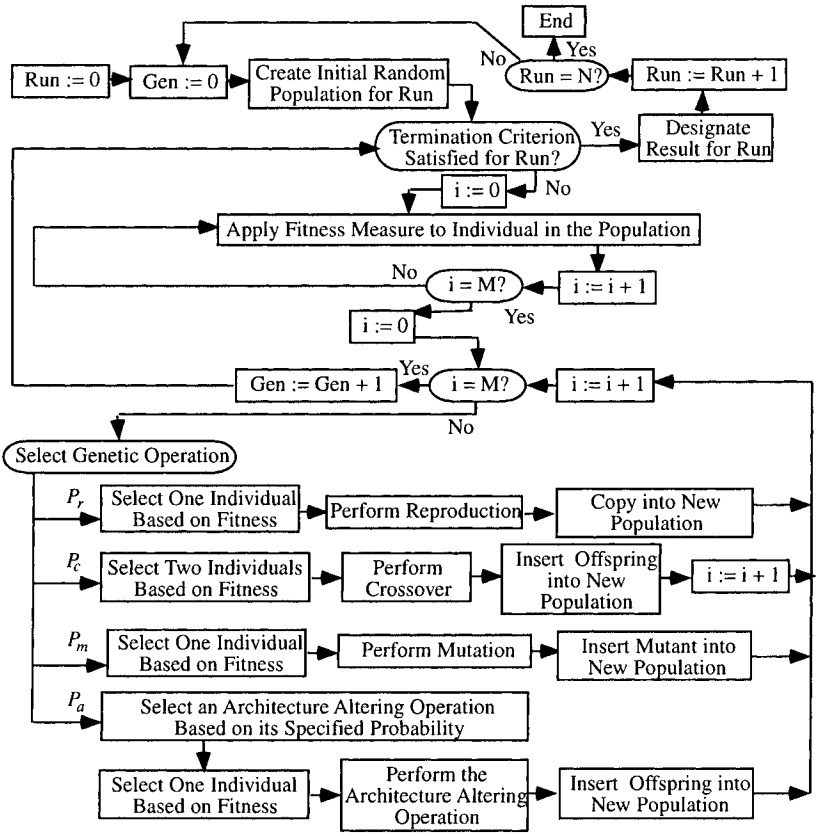

Figure 5.4. Flowchart of genetic programming.

(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 pop-

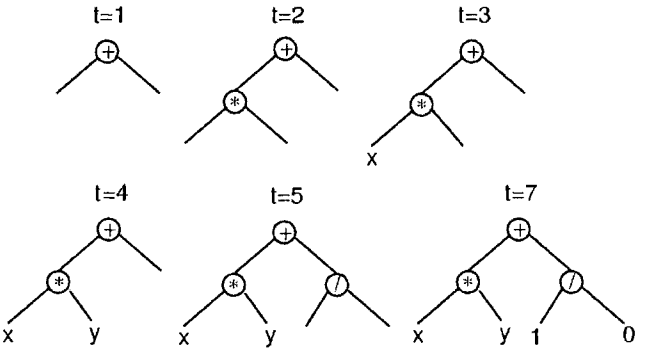

Figure 5.5. Creation of a seven-point tree using the "Full" initialization method (t = time).

ulation 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 5.4 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.

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 5.4) 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). For example, in the *"Full" initialization method* nodes are taken from the function set until a maximum

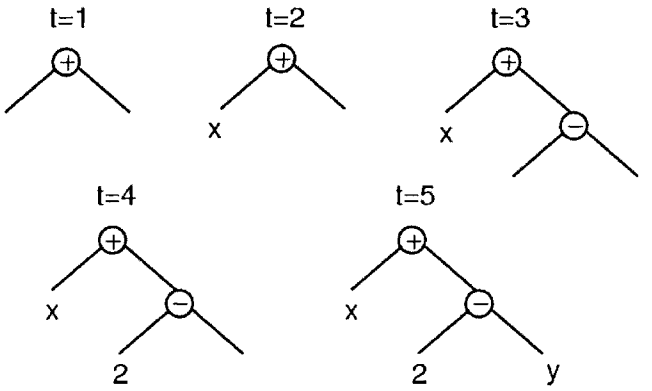

Figure 5.6. Creation of a five-point tree using the "Grow" initialization method (t = time).

tree depth is reached. Beyond that depth only terminals can be chosen. Figure 5.5 shows several snapshots of this process. A variant of this, the *"Grow" initialization method,* allows the selection of nodes from the whole primitive set until the depth limit is reached. Thereafter, it behaves like the "Full" method. Figure 5.6 illustrates this process. Pseudo-code for a recursive implementation of both the "Full" and the "Grow" methods is given in Figure 5.7. The code assumes that programs are represented as prefix-notation expressions.

In general, after the initialization phase, 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 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, this measurement yields a single explicit numerical value, called *fitness.* Normally, fitness evaluation requires executing the programs in the population, often multiple times, *within* the genetic programming system. A variety of execution strategies exist, including the (relatively uncommon) off-line or on-line compilation and linking and the (relatively common) virtual-machine-code compilation and interpretation.

Interpreting a program tree means executing the nodes in the tree in an order that guarantees that nodes are not executed before the value of their arguments (if any) is known. This is usually done by traversing the tree in a recursive way starting from the root node, and postponing the evaluation of each node until the value of its children (arguments) is known. This process is illustrated in Figure 5.8, where the numbers to the right of internal nodes represent the results of evaluating the subtrees rooted at such nodes. In this example, the

```
procedure: gen_rnd_expr
  arguments:
    func_set      /* A function set */
    term_set      /* A terminal set */
    max_d         /* Maximum depth for expressions */
    method        /* Either "Full" or "Grow" */
  results:
    expr          /* An expression in prefix notation */
  begin
    if max_d = 0 or method = "Grow" and random digit = 1 then
      expr = choose_random_element( term_set )
    else
      func = choose_random_element( func_set )
      for i = 1 to arity(func):
        arg_i = gen_rnd_expr(func_set, term_set, max_d - 1, method );
      expr = (func, arg_1, arg_2, ...);
    endif
  end
```

Figure 5.7. Pseudo-code for recursive program generation with the "Full" and "Grow" methods.

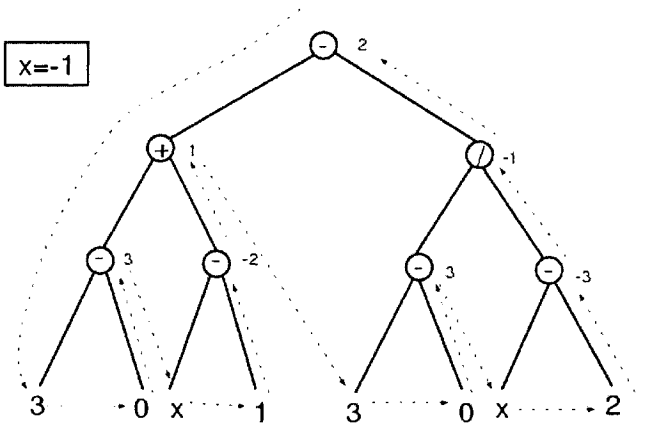

Figure 5.8. Example interpretation of a syntax tree (terminal x is a variable with value -1).

independent variable X evaluates to -1. Figure 5.9 gives a pseudo-code implementation of the interpretation procedure. The code assumes that programs are represented as prefix-notation expressions and that such expressions can be treated as lists of components (where a construct like *expr*(i) can be used to read or set component i of expression *expr*).

Irrespective of the execution strategy adopted, 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 accu-

```
procedure: eval
    arguments:
        expr    /* An expression in prefix notation */
    results:
        value   /* A number */
    begin
        if expr is a list then /* Non-terminal */
            proc = expr(1)
            value = proc(eval(expr(2)),eval(expr(3)),...)
        else /* Terminal */
            if expr is a variable or a constant then
                value = expr
            else /* 0-arity function */
                value = expr()
            endif
        endif
    end
```

Figure 5.9. Typical interpreter for genetic programming.

racy 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 yield both return values and side effects.

The fitness measure is, for many practical problems, multi-objective in the sense that it combines two or more different elements. In practice, the different elements of the fitness measure are 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.

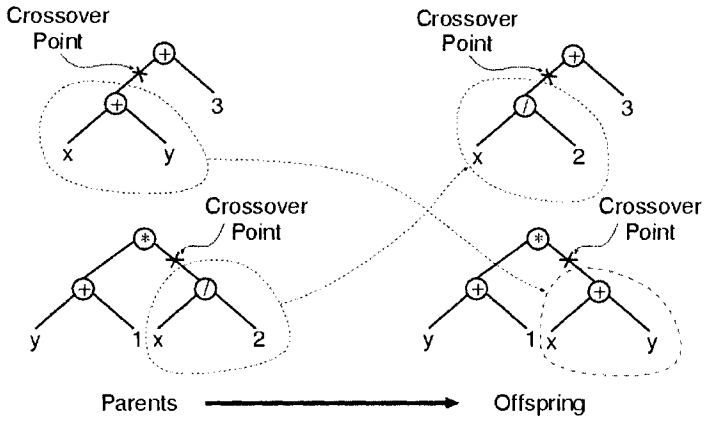

Figure 5.10. Example of two-child crossover between syntax trees.

The genetic operations include crossover (sexual recombination), mutation, reproduction, and the architecture-altering operations. Given copies of two parent trees, typically, *crossover* involves randomly selecting a crossover point (which can equivalently be thought of as either a node or a link between nodes) in each parent tree and swapping the sub-trees rooted at the crossover points, as exemplified in Figure 5.10. Often crossover points are not selected with uniform probability. A frequent strategy is, for example, to select internal nodes (functions) 90% of the times, and any node for the remaining 10% of the times. Traditional *mutation* consists of randomly selecting a mutation point in a tree and substituting the sub-tree rooted there with a randomly generated sub-tree, as illustrated in Figure 5.11. Mutation is sometimes implemented as crossover between a program and a newly generated random program (this is also known as *"headless chicken" crossover*). *Reproduction* involves simply copying certain individuals into the new population. Architecture altering operations will be discussed later in this chapter.

The genetic operations described above 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

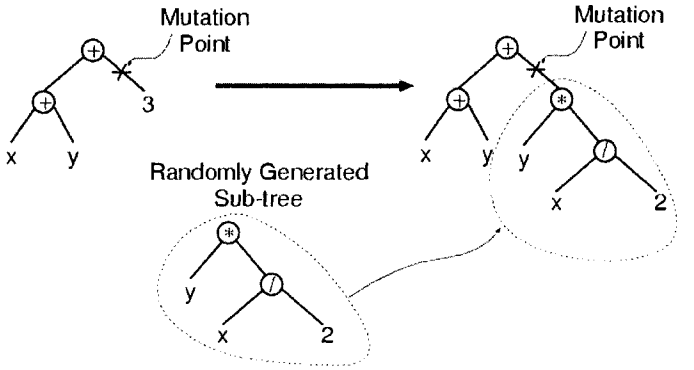

Figure 5.11. Example of sub-tree mutation.

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.

All programs in the initial random population (generation 0) of a run of genetic programming are syntactically valid, executable programs. The genetic operations that are performed during the run (i.e. crossover, mutation, reproduction, and the architecture-altering operations) are designed to produce offspring that are syntactically valid, executable programs. Thus, every individual created during a run of genetic programming (including, in particular, the best-of-run individual) is a syntactically valid, executable program.

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

5.4 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

$$T = \{X, \Re\}$$

Here $\Re$ 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 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 $+$, $-$, $*$, and % functions add, subtract, multiply, and divide, respectively. To avoid run-time errors, the division function % is protected: it 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

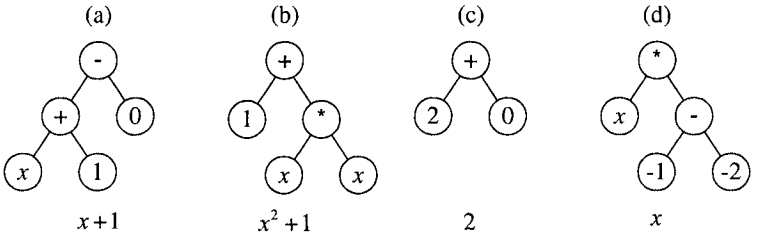

Figure 5.12. Initial population of four randomly created individuals of generation 0.

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 5.4 are now performed.

Genetic programming starts by randomly creating a population of four individual computer programs. The four programs are shown in Figure 5.12 in the form of trees.

The first randomly constructed program tree (Figure 5.12(a)) 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 (+) 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 $x + 1$.

The first program (Figure 5.12(a)) was constructed using the "Grow" method, 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 termi-

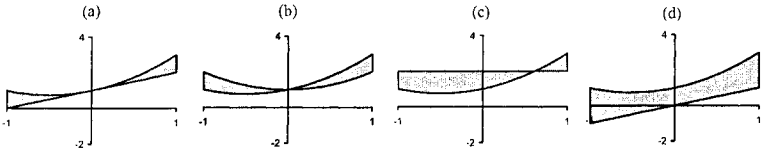

Figure 5.13. The fitness of each of the four randomly created individuals of generation 0 is equal to the area between two curves.

nating 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 5.12(b)) adds the constant terminal 1 to the result of multiplying x by x and is equivalent to x^2+1. The third program (Figure 5.12(c)) adds the constant terminal 2 to the constant terminal 0 and is equivalent to the constant value 2. The fourth program (Figure 5.12(d)) is equivalent to x.

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 5.12 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 5.13 shows (as shaded areas) the integral of the absolute value of the errors between each of the four individuals in Figure 5.12 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 5.13(a)). The integral of absolute error for the parabola $x^2 + 1$ (the second individual) is 1.0 (Figure 5.13(b)). The integrals of the absolute errors for the remaining two individuals are 1.67 (Figure 5.13(c)) and 2.67 (Figure 5.13(d)), respectively.

As can be seen in Figure 5.13, the straight line $x + 1$ (Figure 5.13(a)) 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.

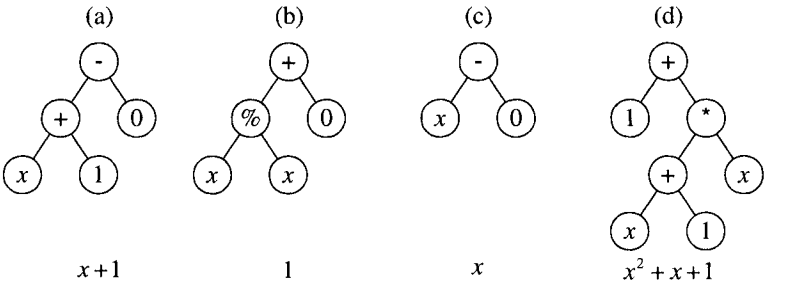

Figure 5.14. Population of generation 1 (after one reproduction, one mutation, and one two-offspring crossover operation).

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 and fitness-proportionate selection. In both methods, 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 5.12(a)) is the most fit individual in the population, it is very likely to be selected to participate in a genetic operation. Let us suppose that this particular individual is, in fact, selected for reproduction. If so, it is copied, without alteration, into the next generation (generation 1). This is shown in Figure 5.14(a) 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 5.12(c)) is selected. One of the three nodes 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 computes the quotient of x and x using the protected division operation %. The resulting individual is shown in Figure 5.14(b). 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. The selection (and 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 5.12(a)), namely the + function, is randomly picked as the crossover point for the first parent. One point of the second parent (Figure 5.12(b)), 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 5.14(c) and 5.14(d). One of the offspring (Figure 5.14(c)) is equivalent to x and is not noteworthy. However, the other offspring (Figure 5.14(d)) 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 5.14(d)) 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 5.14(d)) incorporates a good trait (the quadratic term x^2) from the second parent (Figure 5.12(b)) with two other good traits (the linear term x and constant term of 1) from the first parent (Figure 5.12(a)). 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 $+1$.

5.5 FURTHER 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.

5.5.1 Constrained Syntactic Structures

For certain simple problems (such as the illustrative problem above), 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, 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.

5.5.2 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 useful. 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 parametrized reuse and hierarchical invocation of evolved code. Each ADF resides in a separate function-defining branch within the overall multi-part computer program (see Figure 5.3). When ADFs are being used, a program consists of one (or more) function-defining branches (i.e. ADFs) as well as one or more main result-producing branches. An ADF may possess zero, one, or more dummy variables (formal parameters). The body of an ADF contains its work-performing

steps. Each ADF belongs to a particular program in the population. An ADF may be called by the program's main result-producing branch, another ADF, or another type of branch (such as those described below). Recursion is sometimes allowed. Typically, the ADFs 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 ADF are automatically and simultaneously created during the run of genetic programming.

The program's main result-producing branch and its ADFs typically have different function and terminal sets. A constrained syntactic structure is used to implement ADFs.

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).

5.5.3 Automatically Defined Iterations, Loops, Recursions and Stores

Automatically defined iterations, automatically defined loops, and automatically defined recursions provide means (in addition to ADFs) to reuse code.

Automatically defined stores 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 et al., 1999a).

5.5.4 Program Architecture and Architecture-Altering Operations

The architecture of a program consists of

1 the total number of branches,

2 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),

3 the number of arguments (if any) possessed by each branch, and

4 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:

1 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 previously itemized to six.

2 The run may employ evolutionary selection of the architecture (as described in Koza, 1994a), 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 previously itemized.

3 The run may employ the architecture-altering operations (Koza, 1994c, 1995; Koza et al., 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 previously itemized.

5.5.5 Genetic Programming Problem Solver

The Genetic Programming Problem Solver (GPPS) is described in Koza et al. (1999a, Part 4).

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.

Table 5.1. Eight criteria for saying that an automatically created result is human-competitive.

	Criterion
A	The result was patented as an invention in the past, is an improvement over a patented invention, or would qualify today as a patentable new invention.
B	The result is equal to or better than a result that was accepted as a new scientific result at the time when it was published in a peer-reviewed scientific journal.
C	The result is equal to or better than a result that was placed into a database or archive of results maintained by an internationally recognized panel of scientific experts.
D	The result is publishable in its own right as a new scientific result—independent of the fact that the result was mechanically created.
E	The result is equal to or better than the most recent human-created solution to a long-standing problem for which there has been a succession of increasingly better human-created solutions.
F	The result is equal to or better than a result that was considered an achievement in its field at the time it was first discovered.
G	The result solves a problem of indisputable difficulty in its field.
H	The result holds its own or wins a regulated competition involving human contestants (in the form of either live human players or human-written computer programs).

5.5.6 Developmental Genetic Programming

Developmental genetic programming is used for problems of synthesizing analog electrical circuits, as described in Part 5 of Koza et al. (1999a). When developmental genetic programming is used, a complex structure (such as an electrical circuit) is created from a simple initial structure (the embryo).

5.6 HUMAN-COMPETITIVE RESULTS PRODUCED BY GENETIC PROGRAMMING

Samuel's statement (quoted above) reflects the goal articulated by the pioneers of the 1950s in the fields of artificial intelligence and machine learning, namely to use computers to automatically produce human-like results. Indeed, getting machines to produce human-like results is *the* reason for the existence of the fields of artificial intelligence and machine learning.

Table 5.2. Thirty-six instances of human-competitive results produced by genetic programming.

	Claimed instance	Basis for claim of human-competitiveness	Reference
1	Creation of a better-than-classical quantum algorithm for the Deutsch–Jozsa "early promise" problem	B, F	Spector et al., 1998
2	Creation of a better-than-classical quantum algorithm for Grover's database search problem	B, F	Spector et al., 1999a
3	Creation of a quantum algorithm for the depth-two AND/OR query problem that is better than any previously published result	D	Spector et al., 1999b; Barnum et al., 2000
4	Creation of a quantum algorithm for the depth-one OR query problem that is better than any previously published result	D	Barnum et al., 2000
5	Creation of a protocol for communicating information through a quantum gate that was previously thought not to permit such communication	D	Spector and Bernstein, 2002
6	Creation of a novel variant of quantum dense coding	D	Spector and Bernstein, 2002
7	Creation of a soccer-playing program that won its first two games in the Robo Cup 1997 competition	H	Luke, 1998
8	Creation of a soccer-playing program that ranked in the middle of the field of 34 human-written programs in the Robo Cup 1998 competition	H	Andre and Teller, 1999
9	Creation of four different algorithms for the transmembrane segment identification problem for proteins	B, E	Koza et al., 1994a, 1999
10	Creation of a sorting network for seven items using only 16 steps	A, D	Koza et al., 1999
11	Rediscovery of the Campbell ladder topology for lowpass and highpass filters	A, F	Koza et al., 1999, 2003
12	Rediscovery of the Zobel "M-derived half section" and "constant K" filter sections	A, F	Koza et al., 1999
13	Rediscovery of the Cauer (elliptic) topology for filters	A, F	Koza et al., 1999
14	Automatic decomposition of the problem of synthesizing a crossover filter	A, F	Koza et al., 1999
15	Rediscovery of a recognizable voltage gain stage and a Darlington emitter-follower section of an amplifier and other circuits	A, F	Koza et al., 1999
16	Synthesis of 60 and 96 decibel amplifiers	A, F	Koza et al., 1999
17	Synthesis of analog computational circuits for squaring, cubing, square root, cube root, logarithm, and Gaussian functions	A, D, G	Koza et al., 1999
18	Synthesis of a real-time analog circuit for time-optimal control of a robot	G	Koza et al., 1999
19	Synthesis of an electronic thermometer	A, G	Koza et al., 1999

Table 5.2. (Continued)

	Claimed instance	Basis for claim of human-competitiveness	Reference
20	Synthesis of a voltage reference circuit	A, G	Koza et al., 1999
21	Creation of a cellular automata rule for the majority classification problem that is better than the Gacs–Kurdyumov–Levin rule and all other known rules written by humans	D, E	Koza et al., 1999
22	Creation of motifs that detect the D–E–A–D box family of proteins and the manganese superoxide dismutase family	C	Koza et al., 1999
23	Synthesis of topology for a PID-D2 (proportional, integrative, derivative, and second derivative) controller	A, F	Koza et al., 2003
24	Synthesis of an analog circuit equivalent to Philbrick circuit	A, F	Koza et al., 2003
25	Synthesis of NAND circuit	A, F	Koza et al., 2003
26	Simultaneous synthesis of topology, sizing, placement, and routing of analog electrical circuits	G	Koza et al., 2003
27	Synthesis of topology for a PID (proportional, integrative, and derivative) controller	A, F	Koza et al., 2003
28	Rediscovery of negative feedback	A, E, F, G	Koza et al., 2003
29	Synthesis of a low-voltage balun circuit	A	Koza et al., 2003
30	Synthesis of a mixed analog–digital variable capacitor circuit	A	Koza et al., 2003
31	Synthesis of a high-current load circuit	A	Koza et al., 2003
32	Synthesis of a voltage–current conversion circuit	A	Koza et al., 2003
33	Synthesis of a cubic signal generator	A	Koza et al., 2003
34	Synthesis of a tunable integrated active filter	A	Koza et al., 2003
35	Creation of PID tuning rules that outperform the Ziegler–Nichols and Astrom–Hagglund tuning rules	A, B, D, E, F, G	Koza et al., 2003
36	Creation of three non-PID controllers that outperform a PID controller that uses the Ziegler–Nichols or Astrom–Hagglund tuning rules	A, B, D, E, F, G	Koza et al., 2003

To make the notion of human-competitiveness more concrete, we say that a result is "human-competitive" if it satisfies one or more of the eight criteria in Table 5.1.

As can be seen from Table 5.1, the eight criteria have the desirable attribute of being at arms-length from the fields of artificial intelligence, machine learning, and genetic programming. That is, a result cannot acquire the rating of "human competitive" merely because it is endorsed by researchers *inside* the

specialized fields that are attempting to create machine intelligence. Instead, a result produced by an automated method must earn the rating of "human competitive" independent of the fact that it was generated by an automated method.

Table 5.2 lists the 36 human-competitive instances (of which we are aware) where genetic programming has produced human-competitive results. Each entry in the table is accompanied by the criteria (from Table 5.1) that establish the basis for the claim of human-competitiveness.

There are now 23 instances where genetic programming has duplicated the functionality of a previously patented invention, infringed a previously patented invention, or created a patentable new invention (see criterion A in Table 5.1). Specifically, there are 15 instances where genetic programming has created an entity that either infringes or duplicates the functionality of a previously patented twentieth-century invention, six instances where genetic programming has done the same with respect to an invention patented after January 1st, 2000, and two instances where genetic programming has created a patentable new invention. The two new inventions are general-purpose controllers that outperform controllers employing tuning rules that have been in widespread use in industry for most of the twentieth century.

5.7 SOME PROMISING AREAS FOR FUTURE APPLICATION

Since its early beginnings, the field of genetic and evolutionary computation has produced a cornucopia of results.

Genetic programming and other methods of genetic and evolutionary computation may be especially productive in areas having some or all of the following characteristics:

- where the inter-relationships among the relevant variables are unknown or poorly understood (or where it is suspected that the current understanding may possibly be wrong),
- where finding the size and shape of the ultimate solution to the problem is a major part of the problem,
- where large amounts of primary data requiring examination, classification, and integration are accumulating in computer readable form,
- where there are good simulators to test the performance of tentative solutions to a problem, but poor methods to directly obtain good solutions,
- where conventional mathematical analysis does not, or cannot, provide analytic solutions,
- where an approximate solution is acceptable (or is the only result that is ever likely to be obtained), or

- where small improvements in performance are routinely measured (or easily measurable) and highly prized.

5.8 GENETIC PROGRAMMING THEORY

Genetic programming is a search technique that explores the space of computer programs. As discussed above, the search for solutions to a problem starts from a group of points (random programs) in this search space. Those points that are of above average quality are then used to generate a new generation of points through crossover, mutation, reproduction and possibly other genetic operations. This process is repeated over and over again until a termination criterion is satisfied.

If we could visualize this search, we would often find that initially the population looks a bit like a cloud of randomly scattered points, but that, generation after generation, this cloud changes shape and moves in the search space following a well defined trajectory. Because genetic programming is a stochastic search technique, in different runs we would observe different trajectories. These, however, would very likely show very clear regularities to our eye that could provide us with a deep understanding of how the algorithm is searching the program space for the solutions to a given problem. We could probably readily see, for example, why genetic programming is successful in finding solutions in certain runs and with certain parameter settings, and unsuccessful in/with others.

Unfortunately, it is normally impossible to exactly visualize the program search space due to its high dimensionality and complexity, and so we cannot just use our senses to understand and predict the behavior of genetic programming.

In this situation, one approach to gain an understanding of the behavior of a genetic programming system is to perform many real runs and record the variations of certain numerical descriptors (like the average fitness or the average size of the programs in the population at each generation, the average difference between parent and offspring fitness, etc). Then, one can try to hypothesize explanations about the behavior of the system that are compatible with (and could explain) the empirical observations.

This exercise is very error prone, though, because a genetic programming system is a complex adaptive system with zillions of degrees of freedom. So, any small number of statistical descriptors is likely to be able to capture only a tiny fraction of the complexities of such a system. This is why in order to understand and predict the behavior of genetic programming (and indeed of most other evolutionary algorithms) in precise terms we need to define and then study mathematical models of evolutionary search.

Schema theories are among the oldest, and probably the best-known classes of models of evolutionary algorithms. A *schema* (plural, schemata) is a set of points in the search space sharing some syntactic feature. Schema theories provide information about the properties of individuals of the population belonging to any schema at a given generation in terms of quantities measured at the previous generation, without having to actually run the algorithm.

For example, in the context of genetic algorithms operating on binary strings, a schema is, syntactically, a string of symbols from the alphabet $\{0, 1, *\}$, like *10*1. The character * is interpreted as a "don't care" symbol, so that, semantically, a schema represents a set of bit strings. For example the schema *10*1 represents a set of four strings: {01001, 01011, 11001, 11011}.

Typically, schema theorems are descriptions of how the number (or the proportion) of members of the population belonging to (or matching) a schema varies over time.

For a given schema H the selection/crossover/mutation process can be seen as a Bernoulli trial, because a newly created individual either samples or does not sample H. Therefore, the number of individuals sampling H at the next generation, $m(H, t+1)$ is a binomial stochastic variable. So, if we denote with $\alpha(H, t)$ the success probability of each trial (i.e. the probability that a newly created individual samples H), an *exact schema theorem* is simply

$$E[m(H, t+1)] = M\alpha(H, t)$$

where M is the population size and $E[\,.\,]$ is the expectation operator. Holland's and other approximate schema theories (Holland, 1975; Goldberg, 1989; Whitley, 1994) normally provide a lower bound for $\alpha(H, t)$ or, equivalently, for $E[m(H, t+1)]$. For example, several schema theorems for one-point crossover and point mutation have the following form:

$$\alpha(H, t) \geq p(H, t)(1 - p_m)^{\mathrm{O}(H)} \left[1 - p_c \frac{L(H)}{N-1}\sigma\right]$$

where $m(H, t)$ is number of individuals in the schema H at generation t, M is the population size, $p(H, t)$ is the selection probability for strings in H at generation t, p_m is the mutation probability, $\mathrm{O}(H)$ is the schema order, i.e. number of defining bits, p_c is the crossover probability, $L(H)$ is the defining length, i.e. distance between the furthest defining bits in H, and N is the bitstring length. The factor σ differs in the different formulation of the schema theorem: $\sigma = 1 - m(H, t)/M$ in Holland (1975), where one of the parents was chosen randomly, irrespective of fitness; $\sigma = 1$ in Goldberg (1989); and $\sigma = 1 - p(H, t)$ in Whitley (1994).

More recently, Stephens and Waelbroeck (1997, 1999) have produced exact formulations for $\alpha(H, t)$, which are now known as *"exact" schema theorems* for genetic algorithms. These, however, are beyond the scope of this chapter.

The theory of schemata in genetic programming has had a slow start, one of the difficulties being that the variable size tree structure in genetic programming makes it more difficult to develop a definition of genetic programming schema having the necessary power and flexibility. Several alternatives have been proposed in the literature, which define schemata as composed of one or multiple trees or fragments of trees. Here, however, we will focus only on a particular one, which was proposed by Poli and Langdon (1997, 1998) since this has later been used to develop an exact and general schema theory for genetic programming (Poli and McPhee, 2001; Langdon and Poli, 2002).

In this definition, syntactically, a *genetic programming schema* is a tree with some "don't care" nodes which represents exactly one primitive function or terminal. Semantically, a schema represents all programs that match its size, shape and defining (non-"don't care") nodes. For example, the schema H = (DON'T CARE x(+ y DON'T CARE)) represents the programs (+ x (+ y x)), (+ x (+ y y)), (* x (+ y x)), etc.

The exact schema theorem in Poli and McPhee (2001) gives the expected proportion of individuals matching a schema in the next generation as a function of information about schemata in the current generation. The calculation is non-trivial, but it is easier than one might think.

Let us assume, for simplicity, that only reproduction and (one-offspring) crossover are performed. Because these two operators are mutually exclusive, for a generic schema H we then have

$$\begin{aligned}\alpha(H,t) &= \Pr\left[\text{an individual in } H \text{ is obtained via reproduction}\right] \\ &\quad + \Pr\left[\text{an offspring matching } H \text{ is produced by crossover}\right]\end{aligned}$$

Then, assuming that reproduction is performed with probability p_r and crossover with probability p_c (with $p_r + p_c = 1$), we obtain

$$\begin{aligned}\alpha(H,t) &= p_r \times \Pr\left[\text{an individual in } H \text{ is selected for cloning}\right] \\ &\quad + p_c \Pr\left[\begin{array}{l}\text{the parents and the crossover points} \\ \text{are such that the offspring matches } H\end{array}\right]\end{aligned}$$

Clearly, the first probability in this expression is simply the selection probability for members of the schema H as dictated by, say, fitness-proportionate selection or tournament selection. So,

$$\Pr\left[\text{selecting an individual in } H \text{for cloning}\right] = p(H,t)$$

We now need to calculate the second term in $\alpha(H,t)$: that is, the probability that the parents have shapes and contents compatible with the creation of an offspring matching H, and that the crossover points in the two parents are such that exactly the necessary material to create such an offspring is swapped. This is the harder part of the calculation.

An observation that helps simplify the problem is that, although the probability of choosing a particular crossover point in a parent depends on the actual size and shape of such a parent, the process of crossover point selection is independent from the actual primitives present in the parent tree. So, for example, the probability of choosing any crossover point in the program (+ x (+ y x)) is identical to the probability of choosing any crossover point in the program (AND D1 (OR D1 D2)). This is because the two programs have exactly the same shape. Thanks to this observation we can write

$$\Pr\left[\begin{array}{l}\text{the parents and the crossover points}\\ \text{are such that the offspring matches } H\end{array}\right]$$

$$= \sum_{\substack{\text{For all pairs of}\\ \text{parent shapes } k,\, l}} \sum_{\substack{\text{For all crossover}\\ \text{points } i,\, j \text{ in}\\ \text{shapes } k \text{ and } l}} \Pr\left[\begin{array}{l}\text{Choosing crossover points}\\ i \text{ and } j \text{ in shapes } k \text{ and } l\end{array}\right]$$

$$\times \Pr\left[\begin{array}{l}\text{Selecting parents with shapes } k \text{ and } l, \text{ such that if}\\ \text{crossed at points } i \text{ and } j \text{ produce an offspring in } H\end{array}\right]$$

If, for simplicity, we assume that crossover points are selected with uniform probability, then

$$\Pr\left[\begin{array}{l}\text{Choosing crossover points}\\ i \text{ and } j \text{ in shapes } k \text{ and } l\end{array}\right] = \frac{1}{\text{nodes in shape } k} \times \frac{1}{\text{nodes in shape } l}$$

So, we are left with the problem of calculating the probability of selecting (for crossover) parents having specific shapes while at the same time having an arrangement of primitives such that, if crossed over at certain predefined points, they produce an offspring matching a particular schema of interest.

Again, here we can simplify the problem by considering how crossover produces offspring: it excises a subtree rooted at the chosen crossover point in a parent, and replaces it with a subtree excised from the chosen crossover point in the other parent. This means that the offspring will have the right shape and primitives to match the schema of interest if and only if, after the excision of the chosen subtree, the first parent has shape and primitives compatible with the schema, and the subtree to be inserted has shape and primitives compatible

with the schema. That is,

$$\begin{aligned}
&\Pr\left[\begin{array}{l}\text{Selecting parents with shapes } k \text{ and } l, \text{ such that if}\\ \text{crossed over at points } i \text{ and } j \text{ produce an offspring in } H\end{array}\right]\\
= &\Pr\left[\begin{array}{l}\text{Selecting a root-donating parent with shape } k\\ \text{such that its upper part w.r.t. crossover}\\ \text{point } i \text{ matches the upper part of } H \text{ w.r.t. } i\end{array}\right]\\
&\times \Pr\left[\begin{array}{l}\text{Selecting a subtree-donating parent with shape } l\\ \text{such that its lower part w.r.t. crossover}\\ \text{point } j \text{ matches the lower part of } H \text{ w.r.t. } i\end{array}\right]
\end{aligned}$$

These two selection probabilities can be calculated exactly. However, the calculation requires the introduction of several other concepts and notation, which are beyond the introductory nature of this chapter. These definitions, the complete theory and a number of examples and applications can be found in Poli (2001), Langdon and Poli (2002), and Poli and McPhee (2003a, 2003b).

Although exact schema theoretic models of genetic programming have become available only very recently, they have already started shedding some light on fundamental questions regarding the how and why genetic programming works. Importantly, other important theoretical models of genetic programming have recently been developed which add even more to our theoretical understanding of genetic programming. These, however, go well beyond the scope of this chapter. The interested reader should consult Langdon and Poli (2002) and Poli and McPhee (2003a, 2003b) for more information.

5.9 TRICKS OF THE TRADE

Newcomers to the field of genetic programming often ask themselves (and/or other more experienced genetic programmers) questions such as the following:

1 What is the best way to get started with genetic programming? Which papers should I read?

2 Should I implement my own genetic programming system or should I use an existing package? If so, what package should I use?

Let us start with the first question. A variety of sources of information about genetic programming are available (many of which are listed in the following section). Consulting information available on the Web is certainly a good way to get quick answers for a newcomer who wants to know what genetic programming is. The answer, however, will often be too shallow for someone who really wants to apply genetic programming to solve practical problems. People in this position should probably invest some time going through more detailed

accounts such as Koza (1992), Banzhaf et al. (1998a) and Langdon and Poli (2002), or some of the monographs listed in the following section. Technical papers may be the next stage. The literature on genetic programming is now quite extensive. So, although this is easily accessible thanks to the complete online bibliography listed in the next section, newcomers will often need to be selective in what they read, at least initially. The objective here may be different for different types of readers. Practitioners should probably identify and read only papers which deal with the problem they are interested in. Researchers and Ph.D. students interested in developing a deeper understanding of genetic programming should also make sure they identify and read as many seminal papers as possible, including papers or books on empirical and theoretical studies on the inner mechanisms and behavior of genetic programming. These are frequently cited in other papers and so can easily be identified.

The answer to the second question depends on the particular experience and background of the questioner. Implementing a simple genetic programming system from scratch is certainly an excellent way to make sure one really understands the mechanics of genetic programming. In addition to being an exceptionally useful exercise, this will always result in programmers knowing their systems so well that they will have no problems customizing them for specific purposes (e.g., by adding new, application specific genetic operators, implementing unusual, knowledge-based initialization strategies, etc). All of this, however, requires reasonable programming skills and the will to thoroughly test the resulting system until it fully behaves as expected. If the skills or the time are not available, then the best way to get a working genetic programming application is to retrieve one of the many public-domain genetic programming implementations and adapt this for the user's purposes. This process is faster, and good implementations are often quite robust, efficient, well-documented and comprehensive. The small price to pay is the need to study the available documentation and examples. These often explain also how to modify the genetic programming system to some extent. However, deeper modifications (such as the introduction of new or unusual operators) will often require studying the actual source code of the system and a substantial amount of trial and error. Good, publicly-available GP implementations include LIL-GP from Bill Punch, ECJ from Sean Luke and DGPC from David Andre.

5.10 CONCLUSIONS

In his seminal 1948 paper entitled *Intelligent Machinery*, Turing identified three ways by which human-competitive machine intelligence might be achieved. In connection with one of those ways, Turing (1948) said:

> There is the genetical or evolutionary search by which a combination of genes is looked for, the criterion being the survival value.

Turing did not specify how to conduct the "genetical or evolutionary search" for machine intelligence. In particular, he did not mention the idea of a population-based parallel search in conjunction with sexual recombination (crossover) as described in John Holland's 1975 book *Adaptation in Natural and Artificial Systems*. However, in his 1950 paper *Computing Machinery and Intelligence*, Turing did point out that

> We cannot expect to find a good child-machine at the first attempt. One must experiment with teaching one such machine and see how well it learns. One can then try another and see if it is better or worse. There is an obvious connection between this process and evolution, by the identifications...
>
> Structure of the child machine = Hereditary material
>
> Changes of the child machine = Mutations
>
> Natural selection = Judgment of the experimenter.

That is, Turing perceived in 1948 and 1950 that one possibly productive approach to machine intelligence would involve an evolutionary process in which a description of a computer program (the hereditary material) undergoes progressive modification (mutation) under the guidance of natural selection (i.e. selective pressure in the form of what we now call "fitness").

Today, many decades later, we can see that indeed Turing was right. Genetic programming has started fulfilling Turing's dream by providing us with a systematic method, based on Darwinian evolution, for getting computers to automatically solve hard real-life problems. To do so, it simply requires a high-level statement of what needs to be done (and enough computing power).

Turing also understood the need to evaluate objectively the behavior exhibited by machines, to avoid human biases when assessing their intelligence. This led him to propose an imitation game, now know as the *Turing test for machine intelligence*, whose goals are wonderfully summarized by Arthur Samuel's position statement quoted in the introduction to this chapter.

At present, genetic programming is certainly not in a position to produce computer programs that would pass the full Turing test for machine intelligence, and it might not be ready for this immense task for centuries. Nonetheless, thanks to the constant technological improvements in genetic programming technology, in its theoretical foundations and in computing power, genetic programming has been able to solve tens of difficult problems with human-competitive results (see Table 5.2) in the recent past. These are a small step towards fulfilling Turing and Samuel's dreams, but they are also early signs of things to come. It is, indeed, arguable that in a few years' time genetic programming will be able to *routinely* and *competently* solve important problems for us in a variety of specific domains of application, even when running on a single personal computer, thereby becoming an essential collaborator for many human activities. This, we believe, will be a remarkable step forward towards achieving true, human-competitive machine intelligence.

SOURCES OF ADDITIONAL INFORMATION

Sources of information about genetic programming include the following.

- *Genetic Programming: On the Programming of Computers by Means of Natural Selection* (Koza, 1992) and the accompanying videotape *Genetic Programming: The Movie* (Koza and Rice, 1992).
- *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 et al., 1999a) and the accompanying videotape *Genetic Programming III Videotape: Human-Competitive Machine Intelligence* (Koza et al., 1999b).
- *Genetic Programming IV. Routine Human-Competitive Machine Intelligence* (Koza et al., 2003);
- *Genetic Programming: An Introduction* (Banzhaf et al., 1998a).
- *Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming!* (Langdon, 1998) in the series on genetic programming from Kluwer.
- *Automatic Re-engineering of Software Using Genetic Programming* (Ryan, 1999) in the series on genetic programming from Kluwer.
- *Data Mining Using Grammar Based Genetic Programming and Applications* (Wong and Leung, 2000) in the series on genetic programming from Kluwer.
- *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 et al., 1999a).

- Proceedings of the Genetic Programming Conference (Koza et al., 1996, 1997, 1998).

- 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 et al., 1999; Whitley et al., 2000; Spector et al., 2001; Langdon et al., 2002).

- Proceedings of the Annual Euro-GP Conferences held starting in 1998 (Banzhaf et al., 1998b; Poli et al., 1999, 2000; Miller et al., 2001; Foster et al., 2002).

- Proceedings of the Workshop of Genetic Programming Theory and Practice organized by the Centre for Study of Complex Systems of the University of Michigan (to be published by Kluwer).

- The *Genetic Programming and Evolvable Machines* journal (from Kluwer) started in April 2000.

- Web sites such as www.genetic-programming.org and www.genetic-programming.com.

- LISP code for implementing genetic programming, available in Koza (1992), and genetic programming implementations in other languages such as C, C++, or Java (web sites such as www.genetic-programming.org contain links to computer code in various programming languages).

- 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, 1990) and 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).

- 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).

- Langdon's bibliography 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 of genetic programming and contains over 3034 papers (as of January 2003) and over 880 authors. It provides on-line access to many of the papers.

References

Andre, D. and Teller, A., 1999, Evolving team Darwin United, in: *RoboCup-98: Robot Soccer World Cup II,* M. Asada, and H. Kitano, ed., Lecture Notes in Computer Science, Vol. 1604, Springer, Berlin, pp. 346–352.

Angeline, P. J. and Kinnear Jr, K. E., eds, 1996, *Advances in Genetic Programming 2,* MIT Press, Cambridge, MA.

Babovic, V., 1996, *Emergence, Evolution, Intelligence: Hydroinformatics,* Balkema, Rotterdam.

Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honavar, V., Jakiela, M. and Smith, R. E., eds, 1999, *GECCO-99: Proc. Genetic and Evolutionary Computation Conf.* (Orlando, FL), Morgan Kaufmann, San Mateo, CA.

Banzhaf, W., Nordin, P., Keller, R. E. and Francone, F. D., 1998a, *Genetic Programming: An Introduction,* Morgan Kaufmann, San Mateo, CA.

Banzhaf, W., Poli, R., Schoenauer, M. and Fogarty, T. C., 1998b, *Genetic Programming: Proc. 1st Eur. Workshop* (Paris), Lecture Notes in Computer Science. Vol. 1391, Springer, Berlin.

Barnum, H., Bernstein, H. J., and Spector, L., 2000, Quantum circuits for OR and AND of ORs, *J. Phys. A: Math. Gen.* **33**:8047–8057.

Blickle, T., 1997, *Theory of Evolutionary Algorithms and Application to System Synthesis*, TIK-Schriftenreihe Nr. 17. Zurich, Switzerland: vdf Hochschul, AG an der ETH, Zurich.

Foster, J. A., Lutton, E., Miller, J., Ryan, C. and Tettamanzi, A. G. B., eds, 2002, *Genetic Programming: Proc. 5th Eur. Conf., EuroGP 2002* (Kinsale, Ireland).

Goldberg, D. E., 1989, *Genetic Algorithms in Search, Optimization, and Machine Learning,* Addison-Wesley, Reading, MA.

Holland, J. H., 1975, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence,* University of Michigan Press, Ann Arbor, MI (reprinted 1992, MIT Press, Cambridge, MA).

Iba, H., 1996, *Genetic Programming*, Tokyo Denki University Press, Tokyo, in Japanese.

Jacob, C., 1997, *Principia Evolvica: Simulierte Evolution mit Mathematica,* dpunkt.verlag, Heidelberg.

Jacob, C., 2001, *Illustrating Evolutionary Computation with Mathematica,* Morgan Kaufmann, San Mateo, CA.

Kinnear, K. E. Jr, ed., 1994, *Advances in Genetic Programming*, MIT Press, Cambridge, MA.

Koza, J. R., 1989, Hierarchical genetic algorithms operating on populations of computer programs, in: *Proc. 11th Int. Joint Conf. on Artificial Intelligence,* Vol. 1, Morgan Kaufmann, San Mateo, CA, pp. 768–774.

Koza, J. R., 1990, Genetic programming: a paradigm for genetically breeding populations of computer programs to solve problems, *Stanford University Computer Science Department Technical Report* STAN-CS-90-1314.

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

Koza, J. R., 1994a, *Genetic Programming II: Automatic Discovery of Reusable Programs,* MIT Press, Cambridge, MA.

Koza, J. R., 1994b, *Genetic Programming II Videotape: The Next Generation,* MIT Press, Cambridge, MA.

Koza, J. R., 1994c, Architecture-altering operations for evolving the architecture of a multi-part program in genetic programming, *Stanford University Computer Science Department Technical Report* STAN-CS-TR-94-1528.

Koza, J. R., 1995, Gene duplication to enable genetic programming to concurrently evolve both the architecture and work-performing steps of a computer program, in: *Proc. 14th Int. Joint Conf. on Artificial Intelligence,* Morgan Kaufmann, San Mateo, CA, pp. 734–740.

Koza, J. R., Banzhaf, W., Chellapilla, K., Deb, K., Dorigo, M., Fogel, D. B., Garzon, M. H., Goldberg, D. E., Iba, H. and Riolo, R., eds, 1998, *Genetic Programming 1998: Proc. 3rd Annual Conf.* (Madison, WI), Morgan Kaufmann, San Mateo, CA.

Koza, J. R., Bennett III, F. H, Andre, D. and Keane, M. A., 1999a, *Genetic Programming III: Darwinian Invention and Problem Solving,* Morgan Kaufmann, San Mateo, CA.

Koza, J. R., Bennett III, F. H, Andre, D., Keane, M. A. and Brave, S., 1999b, *Genetic Programming III Videotape: Human-Competitive Machine Intelligence,* Morgan Kaufmann, San Mateo, CA.

Koza, J. R., Deb, K., Dorigo, M., Fogel, D. B., Garzon, M., Iba, H. and Riolo, R. L., eds, *Genetic Programming 1997: Proc. 2nd Annual Conf.* (Stanford University), Morgan Kaufmann, San Mateo, CA.

Koza, J. R., Goldberg, D. E., Fogel, D. B. and Riolo, R. L., eds, 1996, *Genetic Programming 1996: Proc. 1st Annual Conf.* (Stanford University), MIT Press, Cambridge, MA.

Koza, J. R., Keane, M. A., Streeter, M. J., Mydlowec, W., Yu, J. and Lanza, G., 2003, *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*, Kluwer, Dordrecht.

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

Langdon, W. B., 1998, *Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming!* Kluwer, Amsterdam.

Langdon, W. B., Cantu-Paz, E., Mathias, K., Roy, R., Davis, D., Poli, R., Balakrishnan, K., Honavar, V., Rudolph, G., Wegener, J., Bull, L., Potter, M. A., Schultz, A. C., Miller, J. F., Burke, E. and Jonoska, N., eds, 2002, *Proc. 2002 Genetic and Evolutionary Computation Conf.,* Morgan Kaufmann, San Mateo, CA.

Langdon, W. B. and Poli, R., 2002, *Foundations of Genetic Programming,* Springer, Berlin.

Luke, S., 1998, Genetic programming produced competitive soccer softbot teams for RoboCup97, in: *Genetic Programming 1998: Proc. 3rd Annual Conf.* (Madison, WI), J. R. Koza, W. Banzhaf, K. Chellapilla, D. Kumar, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba and R. Riolo, eds, Morgan Kaufmann, San Mateo, CA, pp. 214–222.

Miller, J., Tomassini, M., Lanzi, P. L., Ryan, C., Tettamanzi, A. G. B. and Langdon, W. B., eds, 2001, *Genetic Programming: Proc. 4th Eur. Conf., EuroGP 2001* (Lake Como, Italy), Springer, Berlin.

Nordin, P., 1997, *Evolutionary Program Induction of Binary Machine Code and Its Application,* Krehl, Munster.

Poli, R. and Langdon, W. B., 1997, A new schema theory for genetic programming with one-point crossover and point mutation, in: *Genetic Programming 1997: Proc. 2nd Annual Conf.* (Stanford University), J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba and R. L. Riolo, R. L., eds, Morgan Kaufmann, San Mateo, CA, pp. 278–285.

Poli, R, and Langdon, W. B., 1998, Schema theory for genetic programming with one-point crossover and point mutation, *Evol. Comput.* **6**:231–252.

Poli, R, and McPhee, N. F., 2001, Exact schema theorems for GP with one-point and standard crossover operating on linear structures and their application to the study of the evolution of size, in: *Genetic Programming, Proc. EuroGP 2001, Lake Como, Italy,* J. F. Miller, M. Tomassini, P. L. Lanzi, C. Ryan, A. G. B. Tettamanzi and W. B. Langdon, eds, Lecture Notes in Computer Science, Vol. 2038, Springer, Berlin, pp. 126–142.

Poli, R. and N. F., McPhee, 2003a, General schema theory for genetic programming with subtree-swapping crossover: Part I, *Evol. Comput.* **11**:53–66.

Poli, R. and N. F., McPhee, 2003b, General schema theory for genetic programming with subtree-swapping crossover: Part II, *Evol. Comput.* **11**:169–206.

Poli, R., Nordin, P., Langdon, W. B. and Fogarty, T. C., 1999, *Genetic Programming: Proc. 2nd Eur. Workshop, EuroGP'99,* Lecture Notes in Computer Science. Vol. 1598, Springer, Berlin.

Poli, R., Banzhaf, W., Langdon, W. B., Miller, J., Nordin, P. and Fogarty, T. C., 2000, *Genetic Programming: Proc. Eur. Conf., EuroGP 2000* (Edinburgh), Lecture Notes in Computer Science. Vol. 1802, Springer, Berlin.

Ryan, C., 1999, *Automatic Re-engineering of Software Using Genetic Programming,* Kluwer, Amsterdam.

Samuel, A. L., 1983, AI: Where it has been and where it is going, in: *Proc. 8th Int. Joint Conf. on Artificial Intelligence, Los Altos, CA,* Morgan Kaufmann, San Mateo, CA, pp. 1152–1157.

Spector, L., Barnum, H. and Bernstein, H. J., 1998, Genetic programming for quantum computers, in: *Genetic Programming 1998: Proc. 3rd Annual Conf.* (Madison, WI), J. R. Koza, W. Banzhaf, K. Chellapilla, D. Kumar, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba and R. Riolo, eds, Morgan Kaufmann, San Mateo, CA, pp. 365–373.

Spector, L., Barnum, H. and Bernstein, H. J., 1999a, Quantum computing applications of genetic programming, in: *Advances in Genetic Programming 3,* L. Spector, W. B. Langdon, U.-M. O'Reilly and P. Angeline, eds, MIT Press, Cambridge, MA, pp. 135–160.

Spector, L., Barnum, H., Bernstein, H. J. and Swamy, N., 1999b, Finding a better-than-classical quantum AND/OR algorithm using genetic programming, in: *IEEE Proc. 1999 Congress on Evolutionary Computation,* IEEE, Piscataway, NJ, pp. 2239–2246.

Spector, L. and Bernstein, H. J., 2002, Communication capacities of some quantum gates, discovered in part through genetic programming, in: *Proc. 6th Int. Conf. on Quantum Communication, Measurement, and Computing* (Rinton, Paramus, NJ).

Spector, L., Goodman, E., Wu, A., Langdon, W. B., Voigt, H.-M., Gen, M., Sen, S., Dorigo, M., Pezeshk, S., Garzon, M. and Burke, E., eds, 2001, *Proc. Genetic and Evolutionary Computation Conf., GECCO-2001,* Morgan Kaufmann, San Mateo, CA.

Stephens, C. R. and Waelbroeck, H., 1997, Effective degrees of freedom in genetic algorithms and the block hypothesis, in: *Genetic Algorithms: Proc. 7th Int. Conf.,* Thomas Back, ed., Morgan Kaufmann, San Mateo, CA, pp. 34–40.

Stephens, C. R. and Waelbroeck, H., 1999, Schemata evolution and building blocks, *Evol. Comput.* 7:109–124.

Turing, A. M., 1948, Intelligent machinery. Reprinted in: 1992, *Mechanical Intelligence: Collected Works of A. M. Turing,* D. C. Ince, ed., North-Holland, Amsterdam, pp. 107–127. Also reprinted in: 1969, *Machine Intelligence 5,* B. Meltzer, and D. Michie, ed., Edinburgh University Press, Edinburgh.

Turing, A. M., 1950, Computing machinery and intelligence, *Mind* **59**:433–460. Reprinted in: 1992, *Mechanical Intelligence: Collected Works of A. M. Turing,* D. C. Ince, ed., North-Holland, Amsterdam, pp. 133–160.

Whitley, L. D., 1994, A genetic algorithm tutorial, *Statist. Comput.* **4**:65–85.

Whitley, D., Goldberg, D., Cantu-Paz, E., Spector, L., Parmee, I. and Beyer, H.-G., eds, 2000, *GECCO-2000: Proc. Genetic and Evolutionary Computation Conf.* (Las Vegas, NV), Morgan Kaufmann, San Mateo, CA.

Wong, M. L. and Leung, K. S., 2000, *Data Mining Using Grammar Based Genetic Programming and Applications*, Kluwer, Amsterdam.