Chapter 6
Genetic Programming

Riccardo Poli and John Koza

6.1 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 1983 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:

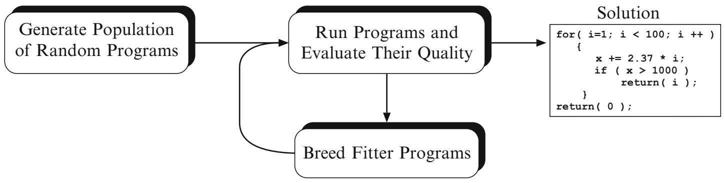
> [T]he aim [is] … to get machines to exhibit behavior, which if done by humans, would be assumed to involve the use of intelligence.

Genetic programming (GP) is a systematic method for getting computers to automatically solve a problem starting from a high-level statement of what needs to be done. GP is a domain-independent method that genetically breeds a population of computer programs to solve a problem. Specifically, GP iteratively transforms a population of computer programs into a new generation of programs by applying analogues of naturally occurring genetic operations. This process is illustrated in Fig. 6.1.

The genetic operations include crossover (sexual recombination), mutation and reproduction. It may also include other analogues of natural operations such as gene duplication, gene deletion and developmental processes which transform an embryo into a fully developed structure. GP is an extension of the genetic algorithm

R. Poli (✉)
School of Computer Science and Electronic Engineering, University of Essex, Colchester, Essex, UK
e-mail: rpoli@essex.ac.uk

J. Koza
Stanford University, Stanford, CA, USA
e-mail: john@johnkoza.com


Generate Population of Random Programs
Run Programs and Evaluate Their Quality
Breed Fitter Programs
Solution

```
for( i=1; i < 100; i ++ )
   {
      x += 2.37 * i;
      if ( x > 1000 )
          return( i );
    }
return( 0 );
```

Fig. 6.1 Main loop of genetic programming

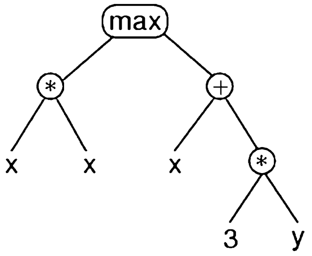


Fig. 6.2 Basic tree-like program representation used in genetic programming

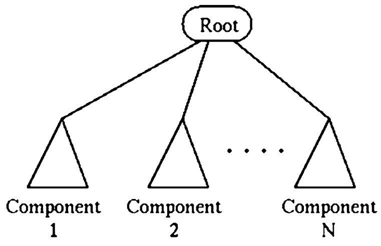


Fig. 6.3 Multi-tree program representation

(Holland 1975) in which the *structures* in the population are not 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 GP as *syntax trees* rather than as lines of code. For example, the simple expression `max(x*x,x+3*y)` is represented as shown in Fig. 6.2. 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 GP, programs can be composed of multiple components (e.g. subroutines). Often in this case the representation used in GP is a set of trees (one for each component) grouped together under a special node called *root*, as illustrated in Fig. 6.3. 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.

GP 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, `max(x*x,x+3*y)` becomes `(max (* x x)(+ x (* 3 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.

6.2 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 GP 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 quality 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 GP 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 GP to automatically program a robot to mop the entire floor of an obstacle-laden room, the human user must tell GP 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 GP 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 GP system. For example, if the goal is to get GP to automatically synthesize an amplifier, the fitness function is the mechanism for telling GP 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. 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.

6.3 Executional Steps of GP

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.

GP 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).

GP iteratively transforms a population of computer programs into a new generation of the population by applying analogues 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 GP.

The executional steps of GP 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:
  - *Reproduction*: Copy the selected individual program into the new population.
  - *Crossover*: Create new offspring program(s) for the new population by recombining randomly chosen parts from two selected programs.
  - *Mutation*: Create one new offspring program for the new population by randomly mutating a randomly chosen part of one selected program.
  - *Architecture-altering operations*: If this feature is enabled, 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 will be a solution (or approximate solution) to the problem.

Figure 6.4 is a flowchart of GP 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 GP system. Once the run is launched, the executional steps as shown in the flowchart (Fig. 6.4) are executed. GP 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 GP (although a human user may exercise judgment as to whether to terminate a run).

GP 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 tree depth is reached. Beyond that depth only terminals can be chosen. Figure 6.5 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.

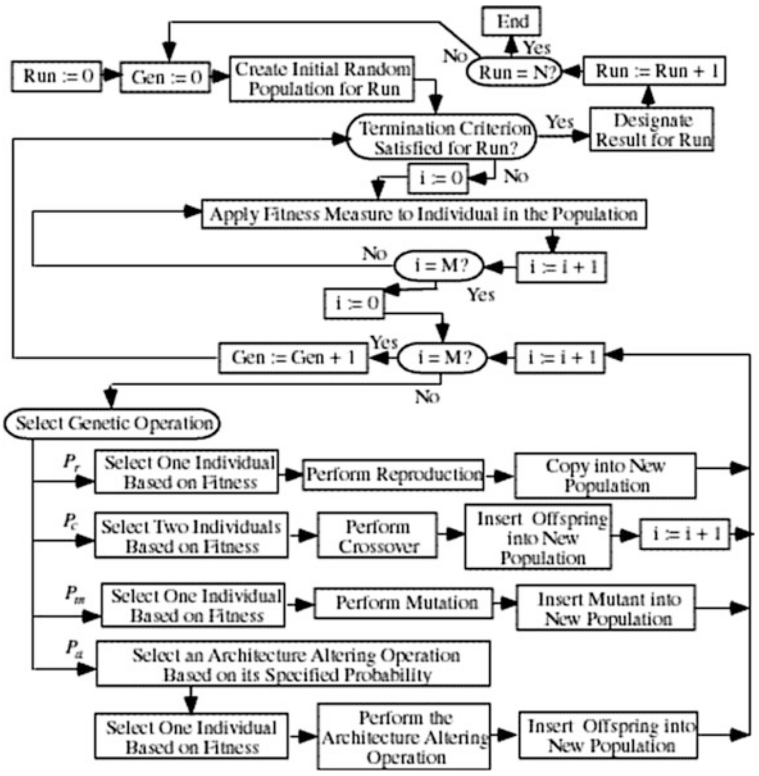

Run := 0
Gen := 0
Create Initial Random Population for Run
Run = N?
End
Yes
No
Run := Run + 1
Designate Result for Run
Termination Criterion Satisfied for Run?
Yes
No
i := 0
Apply Fitness Measure to Individual in the Population
i := i + 1
i = M?
No
Yes
i := 0
Gen := Gen + 1
i = M?
Yes
No
i := i + 1
Select Genetic Operation
P_r
Select One Individual Based on Fitness
Perform Reproduction
Copy into New Population
P_c
Select Two Individuals Based on Fitness
Perform Crossover
Insert Offspring into New Population
i := i + 1
P_m
Select One Individual Based on Fitness
Perform Mutation
Insert Mutant into New Population
P_a
Select an Architecture Altering Operation Based on its Specified Probability
Select One Individual Based on Fitness
Perform the Architecture Altering Operation
Insert Offspring into New Population

Fig. 6.4 Flowchart of genetic programming

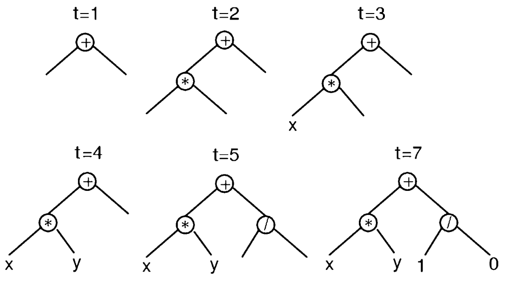


Fig. 6.5 Creation of a seven-point tree using the "Full" initialization method (t = time)

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 evalua-

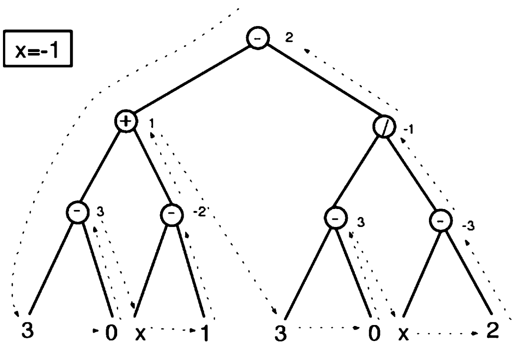


Fig. 6.6 Example interpretation of a syntax tree (the terminal x is a variable holding the value −1)

```
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
```

Fig. 6.7 Typical interpreter for genetic programming

tion requires executing the programs in the population, often multiple times, *within* the GP system. A variety of execution strategies exist. The most common are virtual-machine-code compilation and interpretation. We will look at the latter.

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 Fig. 6.6, where the numbers to the right of internal nodes represent the results of evaluating the subtrees rooted at such nodes. In this example, the independent variable x evaluates to -1. Figure 6.7 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 accuracy of the program

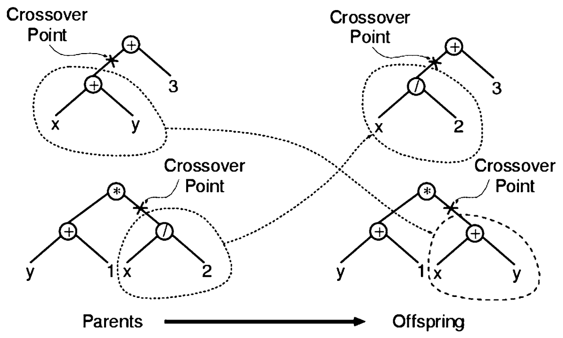


Fig. 6.8 Example of two-child crossover between syntax trees

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) fitter than others. The differences in fitness are then exploited by genetic programming. GP 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 (when they are enabled). Given copies of two parent trees, typically, *crossover* involves randomly selecting a crossover point in each parent tree and swapping the subtrees rooted at the crossover points, as exemplified in Fig. 6.8. Often crossover points are not selected with uniform probability. A frequent strategy is, for example, to select internal nodes (functions) 90 % of the time, 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 subtree rooted there with a randomly generated subtree, as illustrated in Fig. 6.9. *Reproduction* involves simply copying certain individuals into the new population. Architecture-altering operations are discussed later in this chapter.

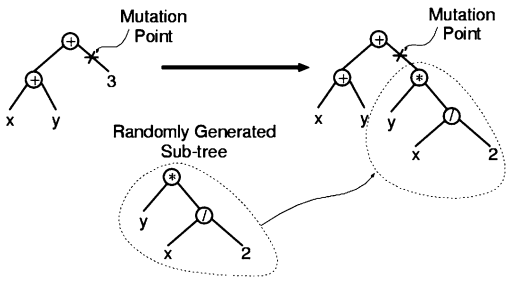


Fig. 6.9 Example of subtree mutation

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 GP 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.

All programs in the initial random population (generation 0) of a run of GP 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 GP (including, in particular, the best-of-run individual) is a syntactically valid, executable program.

There are numerous alternative implementations of GP that vary from the preceding brief description. We will discuss some options in Sect. 6.5.

6.4 Example of a Run of GP

To provide concreteness, this section contains an illustrative run of GP 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 is T= {X, $\Re$}, where $\Re$ 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 for this problem is F= {+, -, *, %}, where 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 GP 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.

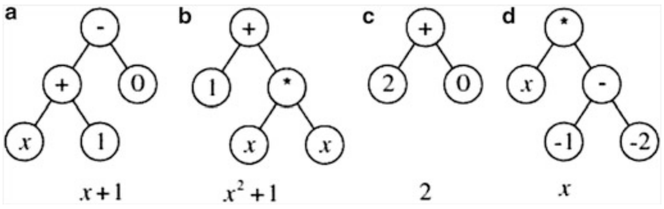


Fig. 6.10 Initial population of four randomly created individuals of generation 0

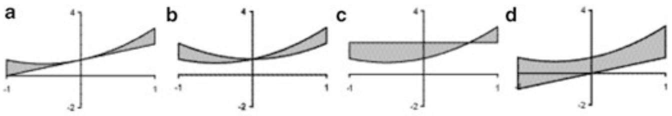


Fig. 6.11 The fitness of each of the four randomly created individuals of generation 0 is equal to the area between two curves

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 GP can be launched. That is, the executional steps shown in the flowchart of Fig. 6.4 are now performed.

GP starts by randomly creating a population of four individual computer programs. The four programs are shown in Fig. 6.10 in the form of trees.

The first randomly constructed program tree (Fig. 6.10a) 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 (Fig. 6.10a) 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 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).

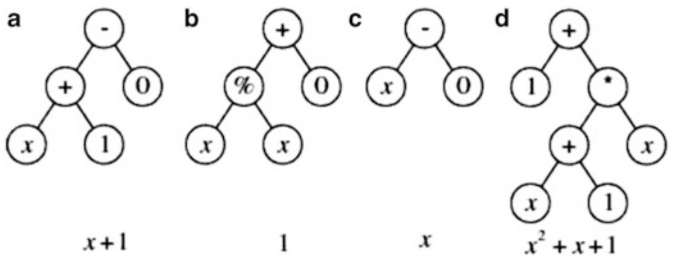


Fig. 6.12 Population of generation 1 (after one reproduction, one mutation, and one two-offspring crossover operation)

The second program (Fig. 6.10b) adds the constant terminal 1 to the result of multiplying x by x and is equivalent to x^2+1. The third program (Fig. 6.10c) adds the constant terminal 2 to the constant terminal 0 and is equivalent to the constant value 2. The fourth program (Fig. 6.10d) 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 Fig. 6.10 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 6.11 shows (as shaded areas) the integral of the absolute value of the errors between each of the four individuals in Fig. 6.10 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 (Fig. 6.11a). The integral of absolute error for the parabola x^2+1 (the second individual) is 1.0 (Fig. 6.11b). The integrals of the absolute errors for the remaining two individuals are 1.67 (Fig. 6.11c) and 2.67 (Fig. 6.11d), respectively.

As can be seen in Fig. 6.11, the straight line $x+1$ (Fig. 6.11a) 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, GP then probabilistically selects relatively fitter 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 is the most fit individual in the population (Fig. 6.10a), 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, 0 is copied, without alteration, into the next generation (generation 1). It is shown in Fig. 6.12a 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 (Fig. 6.10c) 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 Fig. 6.12b. 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 fitter 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 (Fig. 6.10a), namely the + function, is randomly picked as the crossover point for the first parent. One point of the second parent (Fig. 6.10b), 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 Fig. 6.12c, d. One of the offspring (Fig. 6.12c) is equivalent to x and is not noteworthy. However, the other offspring (Fig. 6.12d) 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 (Fig. 6.12d) 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 (Fig. 6.12d) incorporates a good trait (the quadratic term x^2) from the second parent (Fig. 6.10b) with two other good traits (the linear term x and constant term of 1) from the first parent (Fig. 6.10a). 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, GP 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 $+1$.

6.5 Further Features of GP

Various advanced features of GP are not covered by the foregoing illustrative problem and the foregoing discussion of the preparatory and executional steps of GP. In this section we will look at a few alternatives (a more complete and detailed survey is available in Poli et al. 2008).

6.5.1 Automatically Defined Functions and Libraries

Human programmers organize sequences of repeated steps into reusable components such as subroutines, functions and classes. They then repeatedly invoke these components, typically with different inputs. Reuse eliminates the need to "reinvent the wheel" every time a particular sequence of steps is needed. Reuse also makes it possible to exploit a problem's modularities, symmetries and regularities (thereby potentially accelerating the problem-solving process). This can be taken further, as programmers typically organize these components into hierarchies in which top-level components call lower-level ones, which call still lower levels, etc. Automatically defined functions (ADFs) provide a mechanism by which the evolutionary process can evolve these kinds of potentially reusable components. We will review the basic concepts here, but ADFs are discussed in great detail in Koza (1994).

When ADFs are used, a program consists of multiple components. These typically consist of one or more function-defining branches (i.e. ADFs), as well as one or more main result-producing branches (RPBs). The RPB is the "main" program that is executed when the individual is evaluated. It can, however, call the ADFs, which can in turn potentially call each other. A single ADF may be called multiple times by the same RPB, or by a combination of the RPB and other ADFs, allowing the logic that evolution has assembled in that ADF to be re-used in different contexts.

Typically, recursion is prevented by imposing an order on the ADFs within an individual and by restricting calls so that $\texttt{ADF}_i$ can only call $\texttt{ADF}_j$ if $i > j$. Also, in the presence of ADFs, recombination operators are typically constrained to respect the larger structure. That is, during crossover, a subtree from $\texttt{ADF}_i$ can only be swapped with a subtree from another individual's $\texttt{ADF}_i$.

The program's RPB and its ADFs typically have different function and terminal sets. For example, the terminal set for ADFs usually include arguments, such as `arg0` and `arg1`. Typically the user must decide in advance the primitive sets, the number of ADFs and any call restrictions to prevent recursion. These choices can be evolved using the architecture-altering operations described in Sect. 6.5.2.

There have also been proposals for the automated creation of libraries of functions within GP. For example, Angeline and Pollack (1992) and Rosca and Ballard (1996) studied the creation and use of dynamic libraries of subtrees taken from parts of the GP trees in the population.

Naturally, while including ADFs and automatically created libraries makes it possible for modular re-use to emerge, there is no guarantee that they will be used that way. For example, it may be that the RPB never calls an ADF or only calls it once. It is also possible for an ADF to not actually encapsulate any significant logic.

6.5.2 Architecture-Altering Operations

The *architecture* of a program can be defined as the total number of trees, the type of each tree, the number of arguments (if any) possessed by each tree, and, finally, if there is more than one tree, the nature of the hierarchical references (if any) allowed among the trees (e.g. whether $\mathtt{ADF}_1$ can call $\mathtt{ADF}_2$) (Koza 1994).

There are three ways to determine the architecture of the computer programs that will be evolved. Firstly, the user may specify in advance the architecture of the overall program, i.e. perform an *architecture-defining preparatory step* in addition to the five steps itemized in Sect. 6.2. Secondly, a run of GP may employ the *evolutionary design of the architecture* (Koza 1994), thereby enabling the architecture of the overall program to emerge from a competitive process during the run. Finally, a run may employ a set of *architecture-altering operations* (Koza 1994, 1995; Koza et al. 1999a) which, for example, can create, remove or modify ADFs. Note that architecture changes are often designed not to initially change the semantics of the program and, so, the altered program often has exactly the same fitness as its parent. Nevertheless, the new architecture may make it easier to evolve better programs later.

6.5.3 Constraining Structures

Most GP systems require that all subtrees return data of the same type. This ensures that the output of any subtree can be used as one of the inputs to any node. The basic subtree crossover operator shuffles tree components entirely randomly. Type compatibility ensures that crossover cannot lead to incompatible connections between nodes. This is also required to stop mutation from producing illegal programs.

There are cases, however, where this approach is not ideal. For example, there might be constraints on the structure of the acceptable solutions or a problem domain might be naturally represented with multiple types. To apply GP in these cases one needs to be able to use primitives with different type signatures. Below we will look at three approaches to constraining the syntax of the evolved expression trees in GP: simple structure enforcement, strongly typed GP and grammar-based constraints.

If a particular structure is believed or known to be important then one can modify the GP system to require that all individuals have that structure (Koza 1992). Enforcing a user-specified structure on the evolved solutions can be implemented in a number of ways. For example, one can ensure that all the initial individuals have the structure of interest and then constrain crossover and mutation so that they do not alter any of the fixed regions of a tree. An alternative approach is to evolve the various (sub)components separately. A form of constraint-directed search in GP was also proposed in Tsang and Li (2002) and Tsang and Jin (2006).

Since constraints are often driven by or expressed using a type system, a natural approach is to incorporate types and their constraints into the GP system (Montana 1995). In *strongly typed GP*, every terminal has a type, and every function has types for each of its arguments and a type for its return value. The process that generates the initial, random expressions, and all the genetic operators are implemented so as to ensure that they do not violate the type system's constraints. For example, mutation replaces subtrees with new randomly generated trees ensuring that the root of the replacement tree has the same return type as the root of the excised tree. Similarly, crossover only allows the swap of subtrees having the same return type. This basic approach to types can be extended to more complex type systems (Montana 1995; Haynes et al. 1996; Olsson 1994; Yu 2001).

Another natural way to express constraints is via *grammars*, and these have been used in GP in a variety of ways (Whigham 1996; Gruau 1996; Wong and Leung 1996; O'Neill and Ryan 2003; Hoai et al. 2003). In this sort of system, the grammar is typically used to ensure that the initial population is made up of legal programs. The grammar is also used to guide the operations of the genetic operators. Thus we need to keep track not only of the program itself, but also the syntax rules used to derive it.

What actually is evolved in a grammar-based GP system depends on the particular system. Whigham (1996), for example, evolved *derivation trees*, which effectively are a hierarchical representation of which rewrite rules must be applied, and in which order, to obtain a particular program. In this system, crossover is restricted to only swapping subtrees deriving from a common non-terminal symbol in the grammar. The actual program represented by a derivation tree can be obtained by reading out the leaves of the tree one by one from left to right.

Another approach is *grammatical evolution* (GE) which represents individuals as variable-length sequences of integers which are interpreted in the context of a user-supplied grammar (Ryan et al. 1998; O'Neill and Ryan 2003). For each rule in the grammar, the set of alternatives on the right-hand side are numbered from 0 upwards. To create a program from a GE individual one uses the values in the individual to choose which alternative to take in the production rules. If a value exceeds the number of available options it is transformed via a modulus operation.

6.5.4 Developmental GP

By using appropriate terminals, functions and/or interpreters, GP can go beyond the production of computer programs. In *cellular encoding* (Gruau and Whitley 1993; Gruau 1994a,b), programs are interpreted as sequences of instructions which modify (grow) a simple initial structure (embryo). Once the program has finished, the quality of the structure it has produced is measured and this is taken to be the fitness of the program.

Naturally, for cellular encoding to work the primitives of the language must be able to grow structures appropriate to the problem domain. Typical instructions involve the insertion and/or sizing of components, topological modifications of the structure, etc. Cellular encoding GP has successfully been used to evolve neural networks (Gruau and Whitley 1993; Gruau 1994a,b) and electronic circuits (Koza et al. 1996a; Koza et al. 1996b; Koza et al. 1999b), as well as in numerous other domains. A related approach proposed by Hoang et al. (2007) combines tree-adjoining grammars with L-systems (Lindenmayer 1968) to create a system where each stage in the developmental process is a working program that respects the grammatical constraints.

One of the advantages of indirect representations such as cellular encoding is that the standard GP operators can be used to evolve structures (such as circuits) which may have nothing in common with standard GP trees. In many of these systems, the structures being "grown" are also still meaningful (and evaluable) at each point in their development. This allows fitness evaluation. Another important advantage is that structures resulting from developmental processes often have some regularity, which other methods obtain through the use of ADFs, constraints, types, etc.

6.5.5 Probabilistic GP

Genetic programming typically uses an evolutionary algorithm as its main search engine. However, this is not the only option. This section considers work where the exploration is performed by estimation of distribution algorithms (EDAs).

EDAs (Baluja and Caruana 1995; Larrañaga and Lozano 2002) are powerful population-based searchers where the variation operations traditionally implemented via crossover and mutation in EAs are replaced by the process of random sampling from a probability distribution. The distribution is modified generation after generation, using information obtained from the fitter individuals in the population. The objective of these changes in the distribution is to increase the probability of generating individuals with high fitness.

There have been several applications of probabilistic model-based evolution in the areas of tree-based and linear GP. The first EDA-style GP system was effectively an extension of the work in Baluja and Caruana (1995) to trees called probabilistic incremental program evolution (PIPE) (Salustowicz and Schmidhuber 1997). In PIPE, the population is replaced by a hierarchy of probability tables organized into a tree. Each table represents the probability that a particular primitive will be chosen at that specific location in a newly generated program tree. At each generation a population of programs is created based on the current tree of probability tables. Then, the fitness of the new programs is computed and the probability hierarchy is updated on the basis of these fitnesses, so as to make the generation of above-average fitness programs more likely in the next generation. More recent work includes Yanai and Iba (2003), Looks (2007), Looks et al. (2005) and Poli and McPhee (2008b).

A variety of other systems have been proposed which combine the use of grammars and probabilities (Shan et al. 2006). For example, Ratle and Sebag (2001) use a stochastic context-free grammar to generate program trees where the probability of applying each rewrite rule is adapted using an EDA approach. A probabilistic L-system is used by Shan et al. (2003) while a tree-adjunct grammar is used by Abbass et al. (2002) and Shan et al. (2002).

6.5.6 Bloat and Bloat Control

In the early 1990s, researchers began to notice that in addition to progressively increasing their mean and best fitness, GP populations also showed another phenomenon: very often the average size (number of nodes) of the programs in a population after a certain number of generations would start growing at a rapid pace. Typically the increase in program size was not accompanied by any corresponding increase in fitness. This phenomenon is known as *bloat*.

Bloat has significant practical effects: large programs are computationally expensive to evolve and later use, can be hard to interpret, and may exhibit poor generalization. Note that there are situations where one would expect to see program growth as part of the process of solving a problem. For example, GP runs typically start from populations of small random programs, and it may be necessary for the programs to grow in complexity for them to be able to comply with all the fitness cases. So, we should not equate growth with bloat and we should define bloat as *program growth without (significant) return in terms of fitness*.

Numerous empirical techniques have been proposed to control bloat (Langdon et al. 1999; Soule and Foster 1998). In the rest of this section we briefly review some of the most important. In Sect. 6.7 we will review a subset of theoretical explanations for bloat. More information can be found in Poli et al. (2008, 2010).

Rather naturally, the first and simplest method to control code growth is the use of hard limits on the size or depth of the offspring programs generated by the genetic operators. Many implementations of this idea (e.g. Koza 1992) apply a genetic operator and then check whether the offspring is beyond the size or depth limit. If it isn't, the offspring enters the population. If, instead, the offspring exceeds the limit, one of the parents is returned. A problem with this implementation is that parent programs that are more likely to violate the size limit will tend to be copied (unaltered) more often than programs that don't. That is, the population will tend to be filled up with programs that nearly infringe the size limit, which is typically not what is desired.

However, the problem can be fixed by *not returning parents* if the offspring violates a constraint. Instead, one should either return the oversize offspring, but give it a fitness of 0 so that selection will get rid of it at the next generation, or declare the genetic operation failed, and try again.

One can also control bloat by using genetic operators which directly or indirectly have an anti-bloat effect. *Size fair crossover* and *size fair mutation* (Langdon 2000; Crawford-Marks and Spector 2002) achieve this by constraining the choices made during the execution of a genetic operation so as to actively prevent growth. In size-fair crossover, for example, the crossover point in the first parent is selected randomly, as in standard crossover. Then the size of the subtree to be excised is calculated. This is used to constrain the choice of the second crossover point so as to guarantee that the subtree chosen from the second parent will not be "unfairly" big. There are also several *mutation operators* that may help control the average tree size in the population while still introducing new genetic material (e.g. see Kinnear Jr 1993, 1994b; Angeline 1996; Langdon 1998a).

As will be clarified by the size evolution equation presented in Sect. 6.7.2, in systems with symmetric operators, bloat can only happen if there are some longer-than-average programs that are fitter than average or some shorter-than-average programs that are less fit than average, or both. So, it stands to reason that in order to control bloat one needs to somehow modulate the selection probabilities of programs based on their size.

A technique known as the Tarpeian method (Poli 2003) controls bloat by acting directly on selection probabilities. This is done by setting the fitness of randomly chosen longer-than-average programs to 0. This prevents them being parents. By changing how frequently this is done the anti-bloat intensity of Tarpeian control can be modulated. An advantage of the method is that the programs whose fitness is zeroed are never executed, thereby speeding up runs.

The well-known *parsimony pressure method* changes the selection probabilities by subtracting a value based on the size of each program from its fitness (Koza 1992; Zhang and Mühlenbein 1993, 1995; Zhang et al. 1997). Bigger programs have more subtracted and, so, have lower fitness and tend to have fewer children. That is, the new fitness function is $f(x) - c \times \ell(x)$, where $\ell(x)$ is the size of program x, $f(x)$ is its original fitness and c is a constant known as the *parsimony coefficient*. Zhang and Mühlenbein (1995) showed some benefits of adaptively adjusting the coefficient c at each generation but most implementations actually keep the parsimony coefficient constant.

Recently, a theoretically sound method for setting the parsimony coefficient in a principled manner has been proposed (Poli and McPhee 2008a). This is called the *covariant parsimony pressure method*. The method is easy to implement. It recalculates the parsimony coefficient c at each generation using $c = \mathrm{Cov}(\ell, f)/\mathrm{Var}(\ell)$, where $\mathrm{Cov}(\ell, f)$ is the covariance between program size ℓ and program fitness f in the population, and $\mathrm{Var}(\ell)$ is the variance of program sizes. Using this equation ensures that the mean program size remains at the value set by the initialization procedure. There is a variant of the method that allows the user to even decide what function the mean program size should follow over time.

Table 6.1 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 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)

6.6 Human-Competitive Results Produced by GP

Samuel's statement (quoted in Sect. 6.1) 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, gettingmachines to produce human-like results is *the* reason for the existence of the fields of artificial intelligence and machine learning.

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 6.1.

As can be seen from Table 6.1, the eight criteria have the desirable attribute of being at arms-length from the fields of artificial intelligence, machine learning and GP. 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.

Since 2004, a competition has been held annually at ACM's *Genetic and Evolutionary Computation Conference* (termed the Human-Competitive awards—the *Humies*). The $10,000 prize is awarded to projects that have produced automatically-created human-competitive results according to the criteria in Table 6.1. Table 6.2 lists 71 human-competitive instances where GP produced human-competitive results between 1998 and 2009. Each entry in the table is accompanied by the criteria (from Table 6.1) that establish the basis for the claim of human-competitiveness or by the Humies competition where they won a prize or received a honorable mention.

Table 6.2: Seventy-one instances of human-competitive results produced by genetic programming

	Claimed instance	Basis for claim
1	Creation of a better-than-classical quantum algorithm for the Deutsch–Jozsa "early promise" problem (Spector et al. 1998)	B, F
2	Creation of a better-than-classical quantum algorithm for Grover's database search problem (Spector et al. 1999b)	B, F
3	Creation of a quantum algorithm for the depth-two AND/OR query problem that is better than any previously published result (Spector et al. 1999a; Barnum et al. 2000)	D
4	Creation of a quantum algorithm for the depth-one OR query problem that is better than any previously published result (Barnum et al. 2000)	D
5	Creation of a protocol for communicating information through a quantum gate that was previously thought not to permit such communication (Spector and Bernstein 2003)	D
6	Creation of a novel variant of quantum dense coding (Spector and Bernstein 2003)	D
7	Creation of a soccer-playing program that won its first two games in the Robo Cup 1997 competition (Luke 1998)	H
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 (Andre and Teller 1999)	H
9	Creation of four different algorithms for the transmembrane segment identification problem for proteins (Koza 1994, Sects. 18.8 and 18.10; Koza et al. 1999b, Sects. 16.5 and 17.2)	B, E
10	Creation of a sorting network for seven items using only 16 steps (Koza et al. 1999b, Sects. 21.4.4, 23.6, and 57.8.1)	A, D
11	Rediscovery of the Campbell ladder topology for lowpass and highpass filters (Koza et al. 1999b, Sect. 25.15.1; Koza et al. 2003, Sect. 5.2)	A, F
12	Rediscovery of the Zobel "M-derived half section" and "constant K" filter sections (Koza et al. 1999b, Sect. 25.15.2)	A, F
13	Rediscovery of the Cauer (elliptic) topology for filters (Koza et al. 1999b, Sect. 27.3.7)	A, F
14	Automatic decomposition of the problem of synthesizing a crossover filter (Koza et al. 1999b, Sect. 32.3)	A, F
15	Rediscovery of a recognizable voltage gain stage and a Darlington emitter–follower section of an amplifier and other circuits (Koza et al. 1999b, Sect. 42.3)	A, F
16	Synthesis of 60 and 96 decibel amplifiers (Koza et al. 1999b, Sect. 45.3)	A, F

	Claimed instance	Basis for claim
17	Automatic synthesis of asymmetric bandpass filter (Koza et al. 1996b)	
18	Synthesis of analog computational circuits for squaring, cubing, square root, cube root, logarithm, and Gaussian functions (Koza et al. 1999b, Sect. 47.5.3)	A, D, G
19	Synthesis of a real-time analog circuit for time-optimal control of a robot (Koza et al. 1999b, Sect. 48.3)	G
20	Synthesis of an electronic thermometer (Koza et al. 1999b, Sect. 49.3)	A, G
21	Synthesis of a voltage reference circuit (Koza et al. 1999b, Sect. 50.3)	A, G
22	Automatic synthesis of digital-to-analog converter (DAC) circuit (Bennett III et al. 1999)	
23	Automatic synthesis of analog-to-digital (ADC) circuit (Bennett III et al. 1999)	
24	Creation of a cellular automata rule for the majority classification problem that is better than the Gacs–Kurdyumov–Levin (GKL) rule and all other known rules written by humans (Andre et al. 1996; Koza et al. 1999b, Sect. 58.4)	D, E
25	Creation of motifs that detect the D–E–A–D box family of proteins and the manganese superoxide dismutase family (Koza et al. 1999b, Sect. 59.8)	C
26	Synthesis of topology for a PID-D2 (proportional, integrative, derivative, and second derivative) controller (Koza et al. 2003, Sect. 3.7)	A, F
27	Synthesis of an analog circuit equivalent to Philbrick circuit (Koza et al. 2003, Sect. 4.3)	A, F
28	Synthesis of NAND circuit (Koza et al. 2003, Sect. 4.4)	A, F
29	Simultaneous synthesis of topology, sizing, placement, and routing of analog electrical circuits (Koza et al. 2003, Chap. 5)	
30	Synthesis of topology for a PID (proportional, integrative, and derivative) controller (Koza et al. 2003, Sect. 9.2)	A, F
31	Rediscovery of negative feedback (Koza et al. 2003, Chap. 14)	A, E, F, G
32	Synthesis of a low-voltage balun circuit (Koza et al. 2003, Sect. 15.4.1)	A
33	Synthesis of a mixed analog-digital variable capacitor circuit (Koza et al. 2003, Sect. 15.4.2)	A
34	Synthesis of a high-current load circuit (Koza et al. 2003, Sect. 15.4.3)	A
35	Synthesis of a voltage-current conversion circuit (Koza et al. 2003, Sect. 15.4.4)	A

	Claimed instance	Basis for claim
36	Synthesis of a cubic signal generator (Koza et al. 2003, Sect. 15.4.5)	A
37	Synthesis of a tunable integrated active filter (Koza et al. 2003, Sect. 15.4.6)	A
38	Creation of PID tuning rules that outperform the Ziegler–Nichols and Astrom–Hagglund tuning rules (Koza et al. 2003, Chap. 12)	A, B, D, E, F, G
39	Creation of three non-PID controllers that outperform a PID controller that uses the Ziegler–Nichols or Astrom–Hagglund tuning rules (Koza et al. 2003, Chap. 13)	A, B, D, E, F, G
40	An evolved antenna for deployment on NASA's Space Technology 5 Mission (Lohn et al. 2004)	Humies 2004
41	Automatic quantum computer programming: a genetic programming Approach (Spector 2004)	Humies 2004
42	Evolving local search heuristics for SAT using genetic programming (Fukunaga 2004); automated discovery of composite SAT variable-selection heuristics (Fukunaga 2002)	Humies 2004
43	How to draw a straight line using a GP: benchmarking evolutionary design against nineteenth century kinematic synthesis (Lipson 2004)	Humies 2004
44	Organization design optimization using genetic programming (Khosraviani et al. 2004)	Humies 2004
45	Discovery of human-competitive image texture feature programs using genetic programming (Lam and Ciesielski 2004)	Humies 2004
46	Novel image filters implemented in hardware (Sekanina 2003)	Humies 2004
47	Automated re- invention of six patented optical lens systems using genetic programming: two telescope eyepieces, a telescope eyepiece system, an eyepiece for optical instruments, two wide-angle eyepieces, and a telescope eyepiece (Koza et al. 2005, 2008)	Humies 2005
48	Evolution of a human- competitive quantum fourier transform algorithm using genetic programming (Massey et al. 2005)	Humies 2005
49	Evolving assembly programs: how games help microprocessor validation (Corno et al. 2005)	Humies 2005
50	Attaining human-competitive game playing with genetic programming (Sipper 2006); GP-Gammon: using genetic programming to evolve backgammon players (Azaria and Sipper 2005b); GP-Gammon: genetically programming backgammon players (Azaria and Sipper 2005a)	Humies 2005

	Claimed instance	Basis for claim
51	GP-EndChess: using genetic programming to evolve chess endgame (Hauptman and Sipper 2005)	Humies 2005
52	GP-Robocode: using genetic programming to evolve robocode players (Shichel et al. 2005)	Humies 2005
53	Evolving dispatching rules for solving the flexible job-shop problem (Tay and Ho 2008)	Humies 2005
54	Solution of differential equations with genetic programming and the stochastic Bernstein interpolation (Howard and Kolibal 2005)	Humies 2005
55	Determining equations for vegetation-induced resistance using genetic programming (Keijzer et al. 2005)	Humies 2005
56	Sallen–Key filter (Keane et al. 2005)	
57	Using evolution to learn how to perform interest point detection (Trujillo and Olague 2006a); Synthesis of interest point detectors through genetic programming (Trujillo and Olague 2006b)	Humies 2006
58	Evolution of an efficient search algorithm for the mate-in-n problem in chess (Hauptman and Sipper 2007)	Humies 2007
59	Evolving local and global weighting schemes in information retrieval (Cummins and O'Riordan 2006b); An analysis of the solution space for genetically programmed term-weighting schemes in information retrieval (Cummins and O'Riordan 2006a); Term-weighting in information retrieval using genetic programming: a three-stage process (Cummins and O'Riordan 2006c)	Humies 2007
60	Real-time, non-intrusive evaluation of VoIP (Raja et al. 2007)	Humies 2007
61	Automated reverse engineering of nonlinear dynamical systems (Bongard and Lipson 2007)	Humies 2007
62	Genetic programming approach for electron–alkalimetal atom collisions (Radi and El-Bakry 2007; El-Bakry and Radi 2006); Prediction of non-linear system in optics using genetic programming (Radi 2007); Genetic programming approach for flow of steady state fluid between two eccentric spheres (El-Bakry and Radi 2007)	Humies 2007
63	Genetic programming for finite algebras (Spector et al. 2008)	Humies 2008
64	Automatic synthesis of quantum computing circuit for the two-oracle AND/OR problem (Spector and Klein 2008)	
65	Automatic synthesis of quantum computing algorithms for the parity problem a special case of the hidden subgroup problem (Stadelhofer et al. 2008)	

	Claimed instance	Basis for claim
66	Automatic synthesis of mechanical vibration absorbers (Hu et al. 2008)	
67	Automatically finding patches and automated software repair (Nguyen et al. 2009; Weimer et al. 2009)	Humies 2009
68	GP-Rush: using genetic programming to evolve solvers for the rush hour puzzle (Hauptman et al. 2009)	Humies 2009
69	Learning invariant region descriptor operators with genetic programming and the F-measure (Perez and Olague 2008); Evolutionary learning of local descriptor operators for object recognition (Perez and Olague 2009)	Humies 2009
70	Solution of matrix Riccati differential equation for nonlinear singular system using genetic programming (Balasubramaniam and Kumar 2009)	
71	Distilling free-form natural laws from experimental data (Schmidt and Lipson 2009a,b)	

Clearly, Table 6.2 shows GP's potential as a powerful invention machine. There are 31 instances where the human-competitive result produced by GP duplicated the functionality of a previously patented invention, infringed a previously issued patent, or created a patentable new invention. These include one instance where GP has created an entity that either infringes or duplicates the functionality of a previously patented nineteenth-century invention, 21 instances where GP has done the same with respect to previously patented twentieth-century inventions, seven instances where GP has done the same with respect to previously patented twenty-first-century inventions, and two instances where GP 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.

6.7 Genetic Programming Theory

GP 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 GP is a stochastic search technique, in different runs we would observe different trajectories. These, however, would very likely show 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 GP 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 GP.

One approach to gain an understanding of the behavior of a GP system and predict its behavior in precise terms is to define and study mathematical models of evolutionary search. There are a number of cases where this approach has been very successful in illuminating some of the fundamental processes and biases in GP systems. In this section we will review some theoretical approaches to understanding GP. The reader is referred to Langdon and Poli (2002) and Poli et al. (2008, 2010) for more extensive reviews of GP theory.

6.7.1 Models of GP Search

Schema theories are among the oldest and the best known models of evolutionary algorithms (Holland 1992; Whitley 1994). Schema theories are based on the idea of partitioning the search space into subsets, called *schemata*. They are concerned with modeling and explaining the dynamics of the distribution of the population over the schemata. Modern genetic algorithm schema theory (Stephens and Waelbroeck 1997, 1999) provides exact information about the distribution of the population at the next generation in terms of quantities measured at the current generation, without having to actually run the algorithm. Exact schema theories are also available for GP systems with a variety of genetic operators (Poli 2000a,b, 2001a; Langdon and Poli 2002; Poli et al. 2004; Poli and McPhee 2003a,b). Markov chain theory has also started being applied to GP (Poli et al. 2001, 2004; Mitavskiy and Rowe 2006), although so far this hasn't been developed as fully as the schema theory.

Exact mathematical models of GP, such as schema theories and Markov chains, are probabilistic descriptions of the operations of selection, reproduction, crossover and mutation. They explicitly represent how these operations determine which areas of the program space will be sampled by GP, and with what probability. These models treat the fitness function as a black box, however. That is, there is no representation of the fact that in GP, unlike in other evolutionary techniques, the fitness function involves the execution of computer programs on a variety of inputs. In other words, schema theories and Markov chains do not tell us how fitness is distributed in the search space. Yet, without this information, we have no way of closing the loop and fully characterizing the behavior of a GP systems which is always the result of the interaction between the fitness function and the search biases of the representation and genetic operations used in the system.

Fortunately, the characterization of the space of computer programs explored by GP has been another main topic of theoretical research (Langdon and Poli 2002). In this category are theoretical results showing that the distribution of functionality of non-Turing-complete programs approaches a limit as program length increases. That is, although the number of programs of a particular length grows exponentially with length, beyond a certain threshold the fraction of programs implementing any particular functionality is effectively constant. There is a substantial body of empirical evidence indicating that this happens in a variety of systems. In fact, there are also mathematical proofs of these convergence results for two important forms of programs: Lisp (tree-like) S-expressions (without side effects) and machine code programs without loops (Langdon and Poli 2002; Langdon 2002, 2003a,b,c, 2005). Also, recently, Langdon and Poli (2006) and Poli and Langdon (2006) started extending these results to Turing-complete machine code programs.

6.7.2 Bloat

In Sect. 6.5.6 we introduced the notion of bloat and described some effective mechanisms for controlling it. Below we review a subset of theoretical models and explanations for bloat. More information can be found in Poli et al. (2008, 2010) and Langdon and Poli (2002).

There have been some efforts to approximately mathematically model bloat. For example, Banzhaf and Langdon (2002) defined an *executable model of bloat* where only the fitness, the size of active code and the size of inactive code were represented (i.e. there was no representation of program structures). Fitnesses of individuals were drawn from a bell-shaped distribution, while active and inactive code lengths were modified by a size-unbiased mutation operator. Various interesting effects were reported which are very similar to corresponding effects found in GP runs. Rosca (2003) proposed a similar, but slightly more sophisticated model which also included an analogue of crossover.

A strength of these executable models is their simplicity. A weakness is that they suppress or remove many details of the representation and operators typically used in GP. This makes it difficult to verify whether all the phenomena observed in the model have analogues in GP runs, and whether all important behaviors of GP in relation to bloat are captured by the model.

In Poli (2001b) and Poli and McPhee (2003b), a *size evolution equation* for GP was developed, which provided an exact formalization of the dynamics of average program size. The equation has recently been simplified (Poli and McPhee 2008a) giving

$$E[\mu(t+1) - \mu(t)] = \sum_{\ell} \ell \times (p(\ell,t) - \Phi(\ell,t)), \tag{6.1}$$

where $\mu(t+1)$ is the mean size of the programs in the population at generation $t+1$, E is the expectation operator, ℓ is a program size, $p(\ell,t)$ is the probability of selecting programs of size ℓ from the population in generation t, and $\Phi(\ell,t)$ is the proportion of programs of size ℓ in the current generation. The equation applies to a GP system with selection and any form of symmetric subtree crossover. (In a symmetric operator the probability of selecting particular crossover points in the parents does not depend on the order in which the parents are drawn from the population.) Note that the equation constrains what can and cannot happen size-wise in GP populations. Any explanation for bloat has to agree with it.

In particular, Eq. (6.1) shows that there can be bloat only if the selection probability $p(\ell,t)$ is different from the proportion $\Phi(\ell,t)$ for at least some ℓ. So, for bloat to happen there will have to be some small ℓ for which $p(\ell,t) < \Phi(\ell,t)$ and also some bigger ℓ for which $p(\ell,t) > \Phi(\ell,t)$ (at least on average).

We conclude this section with a recent promising explanation for bloat called the *crossover bias theory* (Poli et al. 2007; Dignum and Poli 2007), which is based on and is consistent with Eq. (6.1). The theory goes as follows. On average, each application of subtree crossover removes as much genetic material as it inserts; consequently crossover on its own does not produce growth or shrinkage. While the *mean* program size is unaffected, however, *higher moments* of the distribution are. In particular, crossover pushes the population towards a particular distribution of program sizes, known as a Lagrange distribution of the second kind, where small programs have a much higher frequency than longer ones. For example, crossover generates a very high proportion of single-node individuals. In virtually all problems of practical interest, however, very small programs have no chance of solving the problem. As a result, programs of above average size have a selective advantage over programs of below average size, and the mean program size increases. Because crossover will continue to create small programs, which will then be ignored by selection (in favor of the larger programs), the increase in average size will continue generation by generation.

6.8 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 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 (1950) did point out

> 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. GP 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 known as the Turing test for machine intelligence, whose goals are wonderfully summarized by Arthur Samuel's position statement quoted in the introduction of this chapter.

At present GP 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 GP technology, in its theoretical foundations and in computing power, GP has been able to solve tens of difficult problems with human-competitive results (see Table 6.2) in the recent past. These are a small step towards fulfilling Turing's and Samuel's dreams, but they are also early signs of things to come. It is, indeed, arguable that in a few years' time GP 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 personal computer, thereby becoming an essential collaborator for many of human activities. This, we believe, will be a remarkable step forward towards achieving true, human-competitive machine intelligence.

Tricks of the Trade

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

1. Will GP be able to solve my problem?
2. What is the best way to get started with GP? Which books or papers should I read?
3. Should I implement my own GP system or should I use an existing package? If so, what package should I use?

In the rest of this section we will try to answer the first of these questions by considering the ingredients of successful GP applications, while in the next section we will review some of the wide variety of available sources on GP which should assist readers who wish to explore further.

Based on the experience of numerous researchers over many years, it appears that GP and other evolutionary computation methods have been especially productive in areas having some or all of the following properties:

- The interrelationships among the relevant variables is unknown or poorly understood (or where it is suspected that the current understanding may possibly be wrong).
- Finding the size and shape of the ultimate solution is a major part of the problem.
- Significant amounts of test data are available in computer-readable form.
- There are good simulators to test the performance of tentative solutions to a problem, but poor methods to directly obtain good solutions.
- Conventional mathematical analysis does not, or cannot, provide analytic solutions.
- An approximate solution is acceptable (or is the only result that is ever likely to be obtained).
- Small improvements in performance are routinely measured (or easily measurable) and highly prized.

Sources of Additional Information

Key Books and Journals

There are more than 30 books written in English principally on GP or its applications with more being written. These start with Koza's 1992 book *Genetic Programming* (often referred to as Jaws). Koza has subsequently published three additional books on GP: *Genetic Programming II: Automatic Discovery of Reusable Programs* (1994) deals with ADFs; *Genetic Programming 3* (1999) covers, in particular, the evolution of analog circuits; *Genetic Programming 4* (2003) uses GP for automatic invention. MIT Press published three volumes in the series *Advances in Genetic Programming* (Kinnear Jr 1994a; Angeline and Kinnear Jr 1996; Spector et al. 1999c). The joint GP/genetic algorithms Kluwer book series now contains over 10 books starting with *Genetic Programming and Data Structures* (Langdon 1998b). Apart from Jaws, these tend to be for the GP specialist. The late 1990s saw the introduction of the first textbook dedicated to GP (Banzhaf et al. 1998).

The 2008 book *A Field Guide to Genetic Programming* (Poli et al. 2008) provides a gentle introduction to GP as well as a review of its different flavors and application domains. The book is freely available on the Internet in PDF and HTML formats.

Other titles include Iba (1996), Jacob (1997, 2001), Wong and Leung (2000), Nordin and Johanna (2003), Ryan (1999), Nordin (1997), Blickle (1996), Babovic (1996), Balic (1999), Bhanu et al. (2005), Brabazon and O'Neill (2006), Brezocnik (2000), Chen (2002), Dracopoulos (1997), Eiben and Smith (2003), Krawiec

(2004), Nikolaev and Iba (2006), Riolo and Worzel (2003), Rothlauf (2006), Sekanina (2003) and Spector (2004).

Readers interested in mathematical and empirical analyses of GP behavior may find *Foundations of Genetic Programming* (Langdon and Poli 2002) useful.

Each of Koza's four books has an accompanying video. These videos are now available in DVD format. Also, a small set of videos on specific GP techniques and applications is available via online resources such as Google Video and YouTube.

In addition to GP's own *Genetic Programming and Evolvable Machines* journal, *Evolutionary Computation*, the *IEEE transaction on Evolutionary Computation*, *Complex Systems* (Complex Systems Publication, Inc.), and many others publish GP articles. The GP bibliography (Langdon et al. 1995–2012) lists several hundred different journals worldwide that have published articles related to GP.

GP Implementations

One of the reasons behind the success of GP is that it is easy to implement own versions, and implementing a simple GP system from scratch remains an excellent way to make sure one really understands the mechanics of GP. In addition to being an exceptionally useful exercise, it is often easier to customize (e.g. adding new, application specific genetic operators or implementing unusual, knowledge-based initialization strategies) a system one has built for new purposes than a large GP distribution. All of this, however, requires reasonable programming skills and the will to thoroughly test the resulting system until it behaves as expected.

This is actually an extremely tricky issue in highly stochastic systems such as GP. The problem is that almost any system will produce "interesting" behavior, but it is typically very hard to test whether it is exhibiting the *correct* interesting behavior. It is remarkably easy for small mistakes to go unnoticed for extended periods of time (even years). It is also easy to incorrectly assume that "minor" implementation decisions will not significantly affect the behavior of the system.

An alternative is to use one of the many public domain GP implementations and adapt this for one's purposes. This process is faster, and good implementations are often robust, efficient, well documented and comprehensive. The small price to pay is the need to study the available documentation and examples. These often explain how to modify the GP system to some extent. However, deeper modifications (such as the introduction of new or unusual operators) will often require studying the actual source code and a substantial amount of trial and error. Good publicly available GP implementations include Lil-GP (Punch and Zongker 1998), ECJ (Luke et al. 2000–2013), Open Beagle (Gagné and Parizeau 2002) and GPC++ (Fraser and Weinbrenner 1993–1997). The most prominent commercial implementation remains Discipulus (RML Technologies 1998–2011); see Foster (2001) for a review.

While the earliest GP systems were implemented in Lisp, people have since coded GP in a huge range of different languages, including C/C++, Java, Python, JavaScript, Perl, Prolog, Mathematica, Pop-11, MATLAB, Fortran, Occam and Haskell. Typically, these evolve expressions and programs which look like simplified Lisp. More complex target languages can be supported, however, especially with the use of more advanced tools such as grammars and type systems. Conversely, many successful programs in machine code or low-level languages have also climbed from the primordial ooze of initial randomness.

Online Resources

Online resources appear, disappear, and move with great speed, so the addresses here, which were correct at the time of writing, are obviously subject to change without notice after publication. Hopefully, the most valuable resources should be readily findable using standard search tools.

A key online resource is the GP bibliography (Langdon et al. 1995–2012) available from http://www.cs.bham.ac.uk/~wbl/biblio/. At the time of writing, this bibliography contains over 8,000 GP entries, roughly half of which can be downloaded immediately.

The GP bibliography has a variety of interfaces. It allows for quick jumps between papers linked by authors and allows one to sort the author list by the number of GP publications. Full references are provided in both BibTeX and Refer formats for direct inclusion in papers written in LaTeX and Microsoft Word, respectively. The GP bibliography is also part of the Collection of Computer Sciences Bibliographies (Achilles and Ortyl 1995–2012), which provides a comprehensive Lucerne syntax search engine.

From early on there has been an active, open email discussion list: the GP-list (Genetic Programming Mailing List 2001–2013). The EC-Digest (1985–2013) is a moderated list covering evolutionary computation more broadly, and often contains GP related announcements.

Koza's http://www.genetic-programming.org/ contains a ton of useful information for the novice, including a short tutorial on "What is Genetic Programming" and the Lisp implementation of GP from *Genetic Programming* (Koza 1992).

References

Abbass H, Hoai N, McKay R (2002) AntTAG: a new method to compose computer programs using colonies of ants. In: Proceedings of the CEC 2002, Honolulu, pp 1654–1659

Achilles A-C, Ortyl P (1995–2013) The collection of computer science bibliographies. Avaliable from http://liinwww.ira.uka.de/bibliography/

Andre D, Teller A (1999) Evolving team Darwin united. In: Asada M, Kitano H (eds) RoboCup-98: robot soccer world cup II. LNCS 1604. Springer, Berlin, pp 346–351

Andre D, Bennett FH III, Koza JR (1996) Discovery by genetic programming of a cellular automata rule that is better than any known rule for the majority classification problem. In: Koza JR et al (eds) Proceedings of the 1st annual conference on genetic programming, Stanford. MIT, Cambridge, pp 3–11

Angeline PJ (1996) An investigation into the sensitivity of genetic programming to the frequency of leaf selection during subtree crossover. In: Koza JR et al (eds) Proceedings of the 1st annual conference on genetic programming, Stanford. MIT, Cambridge, pp 21–29

Angeline PJ, Kinnear KE Jr (eds) (1996) Advances in genetic programming 2. MIT, Cambridge

Angeline PJ, Pollack JB (1992) The evolutionary induction of subroutines. In: Proceedings of the 14th annual conference of the cognitive science society. Lawrence Erlbaum, Abingdon, Indiana University, Bloomington, pp 236–241

Azaria Y, Sipper M (2005a) GP-gammon: genetically programming backgammon players. Genet Program Evol Mach 6:283–300. Published online: 12 Aug 2005

Azaria Y, Sipper M (2005b) GP-gammon: using genetic programming to evolve backgammon players. In: Keijzer M et al (eds) Proceedings of the 8th European conference on genetic programming, Lausanne. LNCS 3447. Springer, Berlin, pp 132–142

Babovic V (1996) Emergence, evolution, intelligence; hydroinformatics—a study of distributed and decentralised computing using intelligent agents. AA Balkema, Rotterdam

Balasubramaniam P, Kumar AVA (2009) Solution of matrix Riccati differential equation for nonlinear singular system using genetic programming. Genet Program Evol Mach 10:71–89

Balic J (1999) Flexible manufacturing systems; development–structure–operation–handling–tooling. Manufacturing technology. DAAAM International, Vienna

Baluja S, Caruana R (1995) Removing the genetics from the standard genetic algorithm. In: Prieditis A, Russell S (eds) Proceedings of the 12th international conference on machine learning, Tahoe City. Morgan Kaufmann, San Francisco, pp 38–46

Banzhaf W, Langdon WB (2002) Some considerations on the reason for bloat. Genet Program Evol Mach 3:81–91

Banzhaf W, Nordin P, Keller RE, Francone FD (1998) Genetic programming—an introduction; on the automatic evolution of computer programs and its applications. Morgan Kaufmann, San Francisco

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

Bennett FH III, Koza JR, Keane MA, Yu J, Mydlowec W, Stiffelman O (1999) Evolution by means of genetic programming of analog circuits that perform digital functions. In: Banzhaf W et al (eds) GECCO 1999, Orlando, vol 2. Morgan Kaufmann, San Mateo, pp 1477–1483

Bhanu B, Lin Y, Krawiec K (2005) Evolutionary synthesis of pattern recognition systems. Monographs in computer science. Springer, New York

Blickle T (1996) Theory of evolutionary algorithms and application to system synthesis. PhD thesis, Swiss Federal Institute of Technology, Zurich

Bongard J, Lipson H (2007) Automated reverse engineering of nonlinear dynamical systems. Proc Natl Acad Sci 104:9943–9948

Brabazon A, O'Neill M (2006) Biologically inspired algorithms for financial modelling. Natural computing series. Springer, Berlin

Brameier M, Banzhaf W (2007) Linear genetic programming. Genetic and evolutionary computation series, vol XVI. Springer, Berlin

Brezocnik M (2000) Uporaba genetskega programiranja v inteligentnih proizvodnih sistemih. University of Maribor, Slovenia

Chen S-H (ed) (2002) Genetic algorithms and genetic programming in computational finance. Kluwer, Dordrecht

Corno F, Sanchez E, Squillero G (2005) Evolving assembly programs: how games help microprocessor validation. IEEE Trans Evol Comput 9:695–706

Crawford-Marks R, Spector L (2002) Size control via size fair genetic operators in the PushGP genetic programming system. In: Langdon WB et al (eds) GECCO 2002, New York. Morgan Kaufmann, San Mateo, pp 733–739

Cummins R, O'Riordan C (2006a) An analysis of the solution space for genetically programmed term-weighting schemes in information retrieval. In: Bell DA (ed) AICS 2006, Belfast

Cummins R, O'Riordan C (2006b) Evolving local and global weighting schemes in information retrieval. Inf Retr 9:311–330

Cummins R, O'Riordan C (2006c) Term-weighting in information retrieval using genetic programming: a three stage process. In: Brewka G et al (eds) The 17th European conference on artificial intelligence, Riva del Garda. IOS, Amsterdam, pp 793–794

Dignum S, Poli R (2007) Generalisation of the limiting distribution of program sizes in tree-based genetic programming and analysis of its effects on bloat. In: Thierens D et al (eds) GECCO 2007, London, vol 2. ACM, New York, pp 1588–1595

Dracopoulos DC (1997) Evolutionary learning algorithms for neural adaptive control. Perspectives in neural computing. Springer, Berlin

EC-Digest (1985–2013). Available from http://ec-digest.research.ucf.edu/

Eiben AE, Smith JE (2003) Introduction to evolutionary computing. Springer, Berlin

El-Bakry SY, Radi A (2006) Genetic programming approach for electron–alkali–metal atom collisions. Int J Mod Phys B 20:5463–5471

El-Bakry MY, Radi A (2007) Genetic programming approach for flow of steady state fluid between two eccentric spheres. Appl Rheol 17:68210

Foster JA (2001) Review: discipulus: a commercial genetic programming system. Genet Program Evol Mach 2:201–203

Fraser A, Weinbrenner T (1993–1997) GPC++ genetic programming C++ class library. Available from http://www0.cs.ucl.ac.uk/staff/ucacbbl/ftp/weinbenner/gp.html

Fukunaga A (2002) Automated discovery of composite SAT variable selection heuristics. In: Proceedings of the national conference on artificial intelligence, Edmonton, pp 641–648

Fukunaga AS (2004) Evolving local search heuristics for SAT using genetic programming. In: Deb K et al (eds) GECCO 2004, Seattle. LNCS 3103. Springer, Berlin, pp 483–494

Gagné C, Parizeau M (2002) BEAGLE: a new C++ evolutionary computation framework. In: Langdon WB et al (eds) Proceedings of the GECCO. Morgan Kaufmann, San Mateo, New York, p 888

Genetic Programming Mailing List (2001–2013). Available at http://tech.groups.yahoo.com/group/genetic_programming/

Gruau F (1994a) Neural network synthesis using cellular encoding and the genetic algorithm. PhD thesis, Laboratoire de l'Informatique du Parallilisme, Ecole Normale Superieure de Lyon

Gruau F (1994b) Genetic micro programming of neural networks. In: Kinnear KE Jr (ed) Advances in genetic programming, ch 24. MIT, Cambridge, pp 495–518

Gruau F (1996) On using syntactic constraints with genetic programming. In: Angeline PJ, Kinnear KE Jr (eds) Advances in genetic programming 2, ch 19. MIT, Cambridge, pp 377–394

Gruau F, Whitley D (1993) Adding learning to the cellular development process: a comparative study. Evol Comput 1:213–233

Hauptman A, Sipper M (2005) GP-endchess: using genetic programming to evolve chess endgame players. In: Keijzer M et al (eds) Proceedings of the 8th European conference on genetic programming, Lausanne. LNCS 3447. Springer, Berlin, pp 120–131

Hauptman A, Sipper M (2007) Evolution of an efficient search algorithm for the mate-in-N problem in chess. In: Ebner M et al (eds) Proceedings of the 10th European conference on genetic programming, Valencia. LNCS 4445. Springer, Berlin, pp 78–89

Hauptman A, Elyasaf A, Sipper M, Karmon A (2009) GP-rush: using genetic programming to evolve solvers for the rush hour puzzle. In: Raidl G et al (eds) GECCO 2009, Montreal. ACM, New York, pp 955–962

Haynes TD, Schoenefeld DA, Wainwright RL (1996) Type inheritance in strongly typed genetic programming. In: Angeline PJ, Kinnear KE Jr (eds) Advances in genetic programming 2, ch 18. MIT, Cambridge, pp 359–376

Hoai NX, McKay RI, Abbass HA (2003) Tree adjoining grammars, language bias, and genetic programming. In: Ryan C et al (eds) Proceedings of the EuroGP 2003, Essex. LNCS 2610. Springer, Berlin, pp 335–344

Hoang T-H, Essam D, McKay RI, Nguyen XH (2007) Building on success in genetic programming: adaptive variation and developmental evaluation. In: Proceedings of the 2007 international symposium on intelligent computation and applications, Wuhan. China University of Geosciences Press

Holland JH (1975) Adaptation in natural and artificial systems. University of Michigan Press, Ann Arbor

Holland JH (1992) Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control and artificial intelligence. MIT, Cambridge. First published by University of Michigan Press 1975

Howard D, Kolibal K (2005) Solution of differential equations with genetic programming and the stochastic Bernstein interpolation. Technical report BDS-TR-2005-001, University of Limerick

Hu J, Goodman ED, Li S, Rosenberg R (2008) Automated synthesis of mechanical vibration absorbers using genetic programming. Artif Intell Eng Des Anal Manuf 22:207–217

Iba H (1996) Genetic programming. Tokyo Denki University Press, Tokyo

Jacob C (1997) Principia Evolvica—Simulierte Evolution mit Mathematica. dpunkt.verlag, Heidelberg

Jacob C (2001) Illustrating evolutionary computation with mathematica. Morgan Kaufmann, San Mateo

Keane MA, Koza JR, Streeter MJ (2005) Human-competitive automated engineering design and optimization by means of genetic programming. In: Periaux J et al (eds) Evolutionary algorithms and intelligent tools in engineering optimization. WIT, Southampton

Keijzer M, Baptist M, Babovic V, Uthurburu JR (2005) Determining equations for vegetation induced resistance using genetic programming. In: Beyer H-G et al (eds) GECCO 2005, Washington, DC, vol 2. ACM, New York, pp 1999–2006

Khosraviani B, Levitt RE, Koza JR (2004) Organization design optimization using genetic programming. In: Keijzer M (ed) Late breaking papers at GECCO 2004, Seattle

Kinnear KE Jr (1993) Evolving a sort: lessons in genetic programming. In: Proceedings of the 1993 international conference on neural networks, vol 2. IEEE, Piscataway, San Francisco, CA, pp 881–888

Kinnear KE Jr (ed) (1994a) Advances in genetic programming. MIT, Cambridge

Kinnear KE Jr (1994b) Fitness landscapes and difficulty in genetic programming. In: Proceedings of the 1994 IEEE world conference on computational intelligence, Orlando, vol 1. IEEE, Piscataway, pp 142–147

Koza JR (1992) Genetic programming: on the programming of computers by means of natural selection. MIT, Cambridge

Koza JR (1994) Genetic programming II: automatic discovery of reusable programs. MIT, Cambridge

Koza JR (1995) Two ways of discovering the size and shape of a computer program to solve a problem. In: Eshelman L (ed) Proceedings of the 6th international conference on genetic algorithms, Pittsburgh. Morgan Kaufmann, San Mateo, pp 287–294

Koza JR, Andre D, Bennett FH III, Keane MA (1996a) Use of automatically defined functions and architecture-altering operations in automated circuit synthesis using genetic programming. In: Koza JR et al (eds) Proceedings of the 1st annual conference on genetic programming 1996, Stanford. MIT, Cambridge, pp 132–149

Koza JR, Bennett FH III, Andre D, Keane MA (1996b) Automated WYWIWYG design of both the topology and component values of electrical circuits using genetic programming. In: Koza JR et al (eds) Proceedings of the 1st annual conference on genetic programming 1996, Stanford. MIT, Cambridge, pp 123–131

Koza JR, Bennett FH III, Andre D, Keane MA (1999a) The design of analog circuits by means of genetic programming. In: Bentley P (ed) Evolutionary design by computers, ch 16. Morgan Kaufmann, San Francisco, pp 365–385

Koza JR, Andre D, Bennett FH III, Keane MA (1999b) Genetic programming 3: Darwinian invention and problem solving. Morgan Kaufman, San Mateo

Koza JR, Keane MA, Streeter MJ, Mydlowec W, Yu J, Lanza G (2003) Genetic programming IV: routine human-competitive machine intelligence. Kluwer, Dordrecht

Koza JR, Al-Sakran SH, Jones LW (2005) Automated re-invention of six patented optical lens systems using genetic programming. In: Beyer H-G et al (eds) GECCO 2005, Washington, DC, vol 2. ACM, New York, pp 1953–1960

Koza JR, Al-Sakran SH, Jones LW (2008) Automated ab initio synthesis of complete designs of four patented optical lens systems by means of genetic programming. Artif Intell Eng Des Anal Manuf 22:249–273

Krawiec K (2004) Evolutionary feature programming: cooperative learning for knowledge discovery and computer vision, vol 385. Wydawnictwo Politechniki Poznanskiej, Poznan

Lam B, Ciesielski V (2004) Discovery of human-competitive image texture feature extraction programs using genetic programming. In: Deb K et al (eds) GECCO 2004, Seattle. LNCS 3103. Springer, Berlin, pp 1114–1125

Langdon WB (1998a) The evolution of size in variable length representations. In: IEEE international conference on evolutionary computation, Anchorage. IEEE, Piscataway, pp 633–638

Langdon WB (1998b) Genetic programming and data structures: genetic programming + data structures = automatic programming! Genetic programming, vol 1. Kluwer, Boston

Langdon WB (2000) Size fair and homologous tree genetic programming crossovers. Genet Program Evol Mach 1:95–119

Langdon WB (2002) Convergence rates for the distribution of program outputs. In: Langdon WB et al (eds) GECCO 2002, New York. Morgan Kaufmann, San Mateo, pp 812–819

Langdon WB (2003a) How many good programs are there? How long are they? In: De Jong KA et al (eds) Foundations of genetic algorithms VII. Morgan Kaufmann, San Mateo, pp 183–202

Langdon WB (2003b) Convergence of program fitness landscapes. In: Cantú-Paz E et al (eds) GECCO 2003, Chicago. LNCS 2724. Springer, Berlin, pp 1702–1714

Langdon WB (2003c) The distribution of reversible functions is normal. In: Riolo RL, Worzel B (eds) Genetic programming theory and practise, ch 11. Kluwer, Dordrecht, pp 173–188

Langdon WB (2005) The distribution of amorphous computer outputs. In: Stepney S, Emmott S (eds) The grand challenge in non-classical computation: international workshop, York

Langdon WB, Poli R (2002) Foundations of genetic programming. Springer, Berlin

Langdon WB, Poli R (2006) The halting probability in von Neumann architectures. In: Collet P, Tomassini M, Ebner M et al (eds) Proceedings of the 9th European conference on genetic programming, Budapest. LNCS 3905. Springer, Berlin, pp 225–237

Langdon WB, Gustafson SM, Koza J (1995–2012) The genetic programming bibliography. Available at http://www.cs.bham.ac.uk/~wbl/biblio/

Langdon WB, Soule T, Poli R, Foster JA (1999) The evolution of size and shape. In: Spector L et al (eds) Advances in genetic programming 3, ch 8. MIT, Cambridge, pp 163–190

Larrañaga P, Lozano JA (2002) Estimation of distribution algorithms, a new tool for evolutionary computation. Kluwer, Dordrecht

Lindenmayer A (1968) Mathematic models for cellular interaction in development I and II. J Theor Biol 18:280–299, 300–315

Lipson H (2004) How to draw a straight line using a GP: benchmarking evolutionary design against 19th century kinematic synthesis. In: Keijzer M (ed) Late breaking papers at GECCO 2004, Seattle

Lohn J, Hornby G, Linden D (2004) Evolutionary antenna design for a NASA spacecraft. In: O'Reilly U-M et al (eds) Genetic programming theory and practice II, ch 18. Springer, Berlin, pp 301–315

Looks M (2007) Scalable estimation-of-distribution program evolution. In: Lipson H (ed) GECCO 2007, London. ACM, New York, pp 539–546

Looks M, Goertzel B, Pennachin C (2005) Learning computer programs with the Bayesian optimization algorithm. In: Beyer H-G et al (eds) GECCO 2005, Washington, DC, vol 1. ACM, New York, pp 747–748

Luke S (1998) Genetic programming produced competitive soccer softbot teams for robocup97. In: Koza JR, Banzhaf W, Chellapilla K et al (eds) Proceedings of the 3rd annual conference on genetic programming 1998, Madison. Morgan Kaufmann, San Mateo, pp 214–222

Luke S, Panait L, Balan G et al (2000–2013) ECJ: a java-based evolutionary computation research system. Available at http://cs.gmu.edu/~eclab/projects/ecj/

Massey P, Clark JA, Stepney S (2005) Evolution of a human-competitive quantum Fourier transform algorithm using genetic programming. In: Beyer H-G et al (eds) GECCO 2005, Washington, DC, vol 2. ACM, New York, pp 1657–1663

Mitavskiy B, Rowe J (2006) Some results about the Markov chains associated to GPs and to general EAs. Theor Comput Sci 361:72–110

Montana DJ (1995) Strongly typed genetic programming. Evol Comput 3:199–230

Nguyen TV, Weimer W, Le Goues C, Forrest S (2009) Using execution paths to evolve software patches. In: McMinn P, Feldt R (eds) International conference on software testing, verification and validation workshops, Denver, pp 152–153

Nikolaev N, Iba H (2006) Adaptive learning of polynomial networks genetic programming, backpropagation and Bayesian methods. Genetic and evolutionary computation, vol 4. Springer, Berlin

Nordin P (1997) Evolutionary program induction of binary machine code and its applications. PhD thesis, der Universitat Dortmund am Fachereich Informatik

Nordin P, Johanna W (2003) Humanoider: Sjavlarande robotar och artificiell intelligens. Liber, Stockholm

Olsson JR (1994) Inductive functional programming using incremental program transformation and execution of logic programs by iterative-deepening A* SLD-tree search. PhD thesis, University of Oslo

O'Neill M, Ryan C (2003) Grammatical evolution: evolutionary automatic programming in a arbitrary language. Genetic programming, vol 4. Kluwer, Dordrecht

Perez CB, Olague G (2008) Learning invariant region descriptor operators with genetic programming and the F-measure. In: 19th international conference on pattern recognition, Tampa, pp 1–4

Perez CB, Olague G (2009) Evolutionary learning of local descriptor operators for object recognition. In: Raidl G et al (eds) GECCO 2009, Montreal. ACM, New York, pp 1051–1058

Poli R (2000a) Hyperschema theory for GP with one-point crossover, building blocks, and some new results in GA theory. In: Poli R et al (eds) Proceecings of the EuroGP 2000 on genetic programming, Tübingen. LNCS 1802. Springer, Berlin, pp 163–180

Poli R (2000b) Exact schema theorem and effective fitness for GP with one-point crossover. In: Whitley D et al (eds) GECCO 2000, Las Vegas. Morgan Kaufmann, San Mateo, pp 469–476

Poli R (2001a) Exact schema theory for genetic programming and variable-length genetic algorithms with one-point crossover. Genet Program Evol Mach 2:123–163

Poli R (2001b) General schema theory for genetic programming with subtree-swapping crossover. In: Proceedings of the EuroGP 2001 on genetic programming, Como. LNCS 2038. Springer, Berlin

Poli R (2003) A simple but theoretically-motivated method to control bloat in genetic programming. In: Ryan C et al (eds) Proceedings of the EuroGP 2003 on genetic programming, Essex. LNCS 3003. Springer, Berlin, pp 211–223

Poli R, Langdon WB (2006) Efficient Markov chain model of machine code program execution and halting. In: Riolo RL et al (eds) Genetic programming theory and practice IV. Genetic and evolutionary computation, vol 5, ch 13. Springer, Berlin

Poli R, McPhee NF (2003a) General schema theory for genetic programming with subtree-swapping crossover: I. Evol Comput 11:53–66

Poli R, McPhee NF (2003b) General schema theory for genetic programming with subtree-swapping crossover: II. Evol Comput 11:169–206

Poli R, McPhee NF (2008a) Covariant parsimony pressure in genetic programming. Technical report CES-480, University of Essex

Poli R, McPhee NF (2008b) A linear estimation-of-distribution GP system. In: O'Neill M et al (eds) Proceedings of the EuroGP 2008, Naples. LNCS 4971. Springer, Berlin, pp 206–217

Poli R, Rowe JE, McPhee NF (2001) Markov chain models for GP and variable-length GAs with homologous crossover. In: Spector L et al (eds) GECCO 2001, San Francisco. Morgan Kaufmann, San Mateo, pp 112–119

Poli R, McPhee NF, Rowe JE (2004) Exact schema theory and markov chain models for genetic programming and variable-length genetic algorithms with homologous crossover. Genet Program Evol Mach 5:31–70

Poli R, Langdon WB, Dignum S (2007) On the limiting distribution of program sizes in tree-based genetic programming. In: Ebner M et al (eds) Proceedings of the 10th European conference on genetic programming, Valencia. LNCS 4445. Springer, Berlin, pp 193–204

Poli R, Langdon WB, McPhee NF (2008) A field guide to genetic programming. Published via http://lulu.com and http://www.gp-field-guide.org.uk (with contributions by J. R. Koza)

Poli R, Vanneschi L, Langdon WB, McPhee NF (2010) Theoretical results in genetic programming: the next ten years? Genet Program Evol Mach 11:285–320. 10th anniversary issue: progress in genetic programming and evolvable machines

Punch B, Zongker D (1998) lil-gp genetic programming system. Available at http://garage.cse.msu.edu/software/lil-gp/

Radi A (2007) Prediction of non-linear system in optics using genetic programming. Int J Mod Phys C 18:369–374

Radi AM, El-Bakry SY (2007) Genetic programming approach for positron collisions with alkali-metal atom. In: Thierens D et al (eds) GECCO 2007, London, vol 2. ACM, New York, pp 1756–1756

Raja A, Atif Azad RM, Flanagan C, Ryan C (2007) Real-time, non-intrusive evaluation of voIP. In: Ebner M et al (eds) Proceedings of the 10th European conference on genetic programming, Valencia. LNCS 4445. Springer, Berlin, pp 217–228

Ratle A, Sebag M (2001) Avoiding the bloat with probabilistic grammar-guided genetic programming. In: Collet P et al (eds) Artificial evolution 5th international conference on evolution artificielle, EA, Le Creusot. LNCS 2310. Springer, Berlin, pp 255–266

Riolo RL, Worzel B (eds) (2003) Genetic programming theory and practice. Genetic programming, vol 6. Kluwer, Boston

RML Technologies (1998–2011) Discipulus genetic programming software. Available from http://www.rmltech.com/

Rosca J (2003) A probabilistic model of size drift. In: Riolo RL, Worzel B (eds) Genetic programming theory and practice, ch 8. Kluwer, Dordrecht, pp 119–136

Rosca JP, Ballard DH (1996) Discovery of subroutines in genetic programming. In: Angeline PJ, Kinnear KE Jr (eds) Advances in genetic programming 2, ch 9. MIT, Cambridge, pp 177–202

Rothlauf F (2006) Representations for genetic and evolutionary algorithms, 2nd edn. Springer, Berlin. First published 2002, 2nd edn available electronically

Ryan C (1999) Automatic re-engineering of software using genetic programming. Genetic programming, vol 2. Kluwer, Dordrecht

Ryan C, Collins JJ, O'Neill M (1998) Grammatical evolution: evolving programs for an arbitrary language. In: Banzhaf W et al (eds) Proceedings of the 1st European workshop on genetic programming, Paris. LNCS 1391. Springer, Berlin, pp 83–95

Salustowicz RP, Schmidhuber J (1997) Probabilistic incremental program evolution. Evol Comput 5:123–141

Schmidt M, Lipson H (2009a) Distilling free-form natural laws from experimental data. Science 324:81–85

Schmidt MD, Lipson H (2009b) Solving iterated functions using genetic programming. In: Esparcia AI et al (eds) GECCO 2009 late-breaking papers, Montreal. ACM, New York, pp 2149–2154

Sekanina L (2003) Evolvable components: from theory to hardware implementations. Natural computing. Springer, Berlin

Shan Y, Abbass H, McKay RI, Essam D (2002) AntTAG: a further study. In: Sarker R, McKay B (eds) Proceedings of the 6th Australia–Japan joint workshop on intelligent and evolutionary systems, Canberra

Shan Y, McKay RI, Abbass HA, Essam D (2003) Program evolution with explicit learning: a new framework for program automatic synthesis. In: Sarker R et al (eds) Proceedings of the CEC 2003, Canberra. IEEE, Piscataway, pp 1639–1646

Shan Y, McKay RI, Essam D, Abbass HA (2006) A survey of probabilistic model building genetic programming. In: Pelikan M et al (eds) Scalable optimization via probabilistic modeling: from algorithms to applications. Studies in computational intelligence, vol 33, ch 6. Springer, Berlin, pp 121–160

Shichel Y, Ziserman E, Sipper M (2005) GP-robocode: using genetic programming to evolve robocode players. In: Keijzer M et al (eds) Proceedings of the 8th European conference on genetic programming, Lausanne. LNCS 3447. Springer, Berlin, pp 143–154

Sipper M (2006) Attaining human-competitive game playing with genetic programming. In: El Yacoubi S et al (eds) Proceedings of the 7th international conference on cellular automata, for research and industry, Perpignan. LNCS 4173. Springer, Berlin, p 13. (invited lectures)

Soule T, Foster JA (1998) Effects of code growth and parsimony pressure on populations in genetic programming. Evol Comput 6:293–309

Spector L (2004) Automatic quantum computer programming: a genetic programming approach. Genetic programming, vol 7. Kluwer, Boston

Spector L, Bernstein HJ (2003) Communication capacities of some quantum gates, discovered in part through genetic programming. In: Shapiro JH, Hirota O (eds) Proceedings of the 6th international conference on quantum communication, measurement, and computing, Cambridge. Rinton, Princeton, pp 500–503

Spector L, Klein J (2008) Machine invention of quantum computing circuits by means of genetic programming. Artif Intell Eng Des Anal Manuf 22:275–283

Spector L, Barnum H, Bernstein HJ (1998) Genetic programming for quantum computers. In: Koza JR et al (eds) Proceedings of the 3rd annual conference on genetic programming 1998, Madison. Morgan Kaufmann, San Mateo, pp 365–373

Spector L, Barnum H, Bernstein HJ, Swamy N (1999a) Finding a better-than-classical quantum AND/OR algorithm using genetic programming. In: Angeline PJ et al (eds) Proceedings of the CEC 1999, Washington, DC, vol 3. IEEE, Piscataway, pp 2239–2246

Spector L, Barnum H, Bernstein HJ, Swamy N (1999b) Quantum computing applications of genetic programming. In: Spector L et al (eds) Advances in genetic programming 3, ch 7. MIT, Cambridge, pp 135–160

Spector L, Langdon WB, O'Reilly UM, Angeline PJ (eds) (1999c) Advances in genetic programming 3. MIT, Cambridge

Spector L, Clark DM, Lindsay I, Barr B, Klein J (2008) Genetic programming for finite algebras. In: Keijzer M et al (eds) GECCO 2008, Atlanta. ACM, New York, pp 1291–1298

Stadelhofer R, Banzhaf W, Suter D (2008) Evolving blackbox quantum algorithms using genetic programming. Artif Intell Eng Des Anal Manuf 22:285–297

Stephens CR, Waelbroeck H (1997) Effective degrees of freedom in genetic algorithms and the block hypothesis. In: Bäck T (ed) Proceedings of the 7th international conference on genetic algorithms, East Lansing. Morgan Kaufmann, San Mateo, pp 34–40

Stephens CR, Waelbroeck H (1999) Schemata evolution and building blocks. Evol Comput 7:109–124

Tay JC, Ho NB (2008) Evolving dispatching rules using genetic programming for solving multi-objective flexible job-shop problems. Comput Ind Eng 54:453–473

Trujillo L, Olague G (2006a) Using evolution to learn how to perform interest point detection. In: Tang XY et al (ed) ICPR 2006, Hong Kong, vol 1, pp 211–214

Trujillo L, Olague G (2006b) Synthesis of interest point detectors through genetic programming. In: Keijzer M et al (eds) GECCO 2006, Seattle, vol 1. ACM, New York, pp 887–894

Tsang E, Jin N (2006) Incentive method to handle constraints in evolutionary. In: Collet P et al (eds) Proceedings of the 9th European conference on genetic programming, Budapest. LNCS 3905. Springer, Berlin, pp 133–144

Tsang EPK, Li J (2002) EDDIE for financial forecasting. In: Chen S-H (ed) Genetic algorithms and genetic programming in computational finance, ch 7. Kluwer, Dordrecht, pp 161–174

Turing AM (1948) Intelligent machinery. National Physical Laboratory Report. Reprinted in Ince DC (ed) (1992) Mechanical intelligence: collected works of A. M. Turing, pp 107–127. North-Holland, Amsterdam. Also reprinted in Meltzer B, Michie D (eds) (1969) Machine intelligence 5. Edinburgh University Press

Turing AM (1950) Computing machinery and intelligence. Mind 49:433–460

Weimer W, Nguyen T, Le Goues C, Forrest S (2009) Automatically finding patches using genetic programming. In: Fickas S (ed) International conference on software engineering, Vancouver, pp 364–374

Whigham PA (1996) Search bias, language bias, and genetic programming. In: Koza JR et al (eds) Proceedings of the 1st annual conference on genetic programming 1996, Stanford. MIT, Cambridge, pp 230–237

Whitley LD (1994) A genetic algorithm tutorial. Stat Comput 4:65–85

Wong ML, Leung KS (1996) Evolving recursive functions for the even-parity problem using genetic programming. In: Angeline PJ, Kinnear KE Jr (eds) Advances in genetic programming 2, ch 11. MIT, Cambridge, pp 221–240

Wong ML, Leung KS (2000) Data mining using grammar based genetic programming and applications. Genetic programming, vol 3. Kluwer, Dordrecht

Yanai K, Iba H (2003) Estimation of distribution programming based on bayesian network. In: Sarker R et al (eds) Proceedings of the CEC 2003, Canberra. IEEE, Piscataway, pp 1618–1625

Yu T (2001) Hierachical processing for evolving recursive and modular programs using higher order functions and lambda abstractions. Genet Program Evol Mach 2:345–380

Zhang B-T, Mühlenbein H (1993) Evolving optimal neural networks using genetic algorithms with Occam's razor. Complex Syst 7:199–220

Zhang B-T, Mühlenbein H (1995) Balancing accuracy and parsimony in genetic programming. Evol Comput 3:17–38

Zhang B-T, Ohm P, Mühlenbein H (1997) Evolutionary induction of sparse neural trees. Evol Comput 5:213–236