Genetic Programming: An Introduction and Tutorial, with a Survey of Techniques and Applications

William B. Langdon[1], Riccardo Poli[2], Nicholas F. McPhee[3], and John R. Koza[4]

[1] Departments of Biological and Mathematical Sciences, University of Essex, UK, wlangdon@essex.ac.uk
[2] Department of Computing and Electronic Systems, University of Essex, UK, rpoli@essex.ac.uk
[3] Division of Science and Mathematics, University of Minnesota, Morris, USA, mcphee@morris.umn.edu
[4] Stanford University, Stanford, CA, USA, john@johnkoza.com

1 Introduction

The goal of having computers automatically solve problems is central to artificial intelligence, machine learning, and the broad area encompassed by what Turing called 'machine intelligence' [384]. Machine learning pioneer Arthur Samuel, in his 1983 talk entitled 'AI: Where It Has Been and Where It Is Going' [337], stated that the main goal of the fields of machine learning and artificial intelligence 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 an evolutionary computation (EC) technique that automatically solves problems without having to tell the computer explicitly how to do it. At the most abstract level GP is a *systematic, domain-independent* method for getting computers to *automatically* solve problems starting from a *high-level statement* of what needs to be done.

Over the last decade, GP has attracted the interest of streams of researchers around the globe. This Chapter is intended to give an overview of the basics of GP, to summarize important work that gave direction and impetus to research in GP as well as to discuss some interesting new directions and applications. Things change fast in this field, as investigators discover new ways of doing things, and new things to do with GP. It is impossible to cover all aspects of this area, even within the generous page limits of this chapter. Thus this

W.B. Langdon et al.: *Genetic Programming: An Introduction and Tutorial, with a Survey of Techniques and Applications*, Studies in Computational Intelligence (SCI) **115**, 927–1028 (2008)
www.springerlink.com

Algorithm 1 Abstract GP algorithm

1: Randomly create an *initial population* of programs from the available primitives (see Sect. 2.2).
2: **repeat**
3: *Execute* each program and ascertain its fitness.
4: *Select* one or two program(s) from the population with a probability based on fitness to participate in genetic operations (see Sect. 2.3).
5: Create new individual program(s) by applying *genetic operations* with specified probabilities (see Sect. 2.4).
6: **until** an acceptable solution is found or some other stopping condition is met (for example, reaching a maximum number of generations).
7: **return** the best-so-far individual.

Chapter should be seen as a snapshot of the view we, the authors, have at the time of writing.

1.1 GP in a Nutshell

Technically, GP is a special evolutionary algorithm (EA) where the individuals in the population are *computer programs*. So, generation by generation GP *iteratively* transforms populations of programs into other populations of programs as illustrated in Fig. 1. During the process, GP constructs new programs by applying genetic operations which are specialized to act on computer programs.

Algorithmically, GP comprises the steps shown in Algorithm 1. The main genetic operations involved in GP (line 5 of Algorithm 1) are the following:

- **Crossover:** the creation of one or two offspring programs by recombining randomly chosen parts from two selected programs.
- **Mutation:** the creation of one new offspring program by randomly altering a randomly chosen part of one selected program.

Some GP systems also support structured solutions (see, for example, Sect. 5.1), and some of these then include *architecture-altering operations*

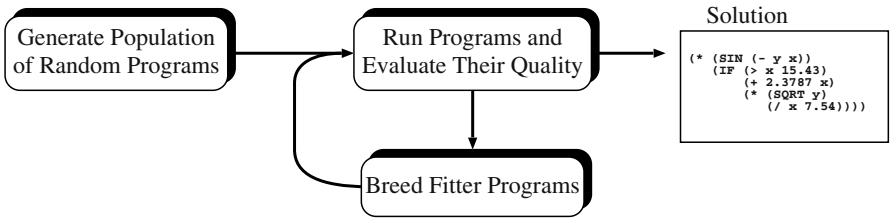

Fig. 1. GP main loop

which randomly alter the architecture (for example, the number of subroutines) of a program to create a new offspring program. Also, often, in addition of crossover, mutation and the architecture-altering operations, an operation which simply copies selected individuals in the next generation is used. This operation, called *reproduction*, is typically applied only to produce a fraction of the new generation.

1.2 Overview of the Chapter

This Chapter starts with an overview of the key representations and operations in GP (Sect. 2), a discussion of the decisions that need to be made before running GP (Sect. 3), and an example of a GP run (Sect. 4).

This is followed by descriptions of some more advanced GP techniques including: automatically defined functions (Sect. 5.1) and architecture-altering operations (Sect. 5.2), the GP problem solver (Sect. 5.3), systems that constrain the syntax of evolved programs in some way (for instance, using grammars or type systems; Sect. 5.4) and developmental GP (Sect. 5.5). Alternative program representations, namely linear GP (Sect. 6.1) and graph-based GP (Sect. 6.2) are then discussed.

After this survey of representations, we provide a review of the enormous variety of applications of GP, including curve fitting and data modeling (Sect. 7.1), human competitive results (Sect. 7.2) and much more, and a substantial collection of 'tricks of the trade' used by experienced GP practitioners (Sect. 8). We also give an overview of some of the considerable work that has been done on the theory of GP (Sect. 9).

After concluding the Chapter (Sect. 10), we provide a resources appendix that reviews the many sources of further information on GP, its applications, and related problem solving systems.

2 Representation, Initialization and Operators in Tree-Based GP

In this Section we will introduce the basic tools and terms used in genetic programming. In particular, we will look at how solutions are represented in most GP systems (Sect. 2.1), how one might construct the initial, random population (Sect. 2.2), and how selection (Sect. 2.3) as well as recombination and mutation (Sect. 2.4) are used to construct new individuals.

2.1 Representation

In GP programs are usually expressed as *syntax trees* rather than as lines of code. Figure 2 shows, for example, the tree representation of the program `max(x*x,x+3*y)`. Note how the variables and constants in the program

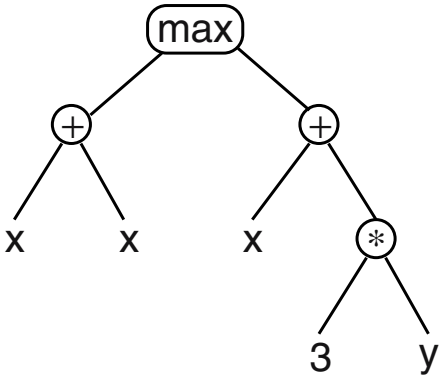

Fig. 2. GP syntax tree representing `max(x*x,x+3y)`

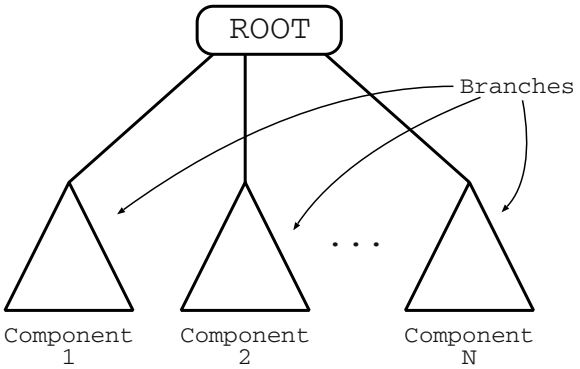

Fig. 3. Multi-component program representation

(`x`, `y`, and `3`), called *terminals* in GP, are leaves of the tree, while the arithmetic operations (`+`, `*`, and `max`) are internal nodes (typically called *functions* in the GP literature). The sets of allowed functions and terminals together form the *primitive set* of a GP system.

In more advanced forms of GP, programs can be composed of multiple components (say, subroutines). In this case the representation used in GP is a set of trees (one for each component) grouped together under a special root node that acts as glue, as illustrated in Fig. 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.

It is common in the GP literature to represent expressions in a *prefix* notation similar to that used in Lisp or Scheme. For example, `max(x*x,x+3*y)` becomes `(max (* x x) (+ x (* 3 y)))`. This notation often makes it easier to see the relationship between (sub)expressions and their corresponding

(sub)trees. Therefore, in the following, we will use trees and their corresponding prefix-notation expressions interchangeably.

How one implements GP trees will obviously depend a great deal on the programming languages and libraries being used. Most traditional languages used in AI research (such as Lisp and Prolog), many recent languages (say Ruby and Python), and the languages associated with several scientific programming tools (namely, MATLAB® and Mathemtica®) provide automatic garbage collection and dynamic lists as fundamental data types making it easy to directly implement expression trees and the necessary GP operations. In other languages one may have to implement lists/trees or use libraries that provide such data structures.

In high performance environments, however, the tree-based representation may be too memory-inefficient since it requires the storage and management of numerous pointers. If all functions have a fixed arity (which is extremely common in GP applications) the brackets become redundant in prefix-notation expressions, and the tree can be represented as a simple linear sequence. For example, the expression `(max (* x x) (+ x (* 3 y)))` could be written unambiguously as the sequence `max * x x + x * 3 y`. The choice of whether to use such a linear representation or an explicit tree representation is typically guided by questions of convenience, efficiency, the genetic operations being used (some may be more easily or more efficiently implemented in one representation), and other data one may wish to collect during runs (for instance, it is sometimes useful to attach additional information to nodes, which may require that they be explicitly represented). There are also numerous high-quality, freely available GP implementations (see the resources in the appendix at the end of this chapter for more information).

While these tree representations are the most common in GP, there are other important representations, some of which are discussed in Sect. 6.

2.2 Initializing the Population

Similar to other evolutionary algorithms, in GP the individuals in the initial population are randomly generated. There are a number of different approaches to generating this random initial population. Here we will describe two of the simplest (and earliest) methods (the *Full* and *Grow* methods), and a widely used combination of the two known as *Ramped half-and-half*.

In both the *Full* and *Grow* methods, the initial individuals are generated subject to a pre-established maximum depth. In the *Full* method (so named because it generates full trees) nodes are taken at random from the function set until this maximum tree depth is reached, and beyond that depth only terminals can be chosen. Figure 4 shows snapshots of this process in the construction of a full tree of depth 2. The children of the * node, for example,

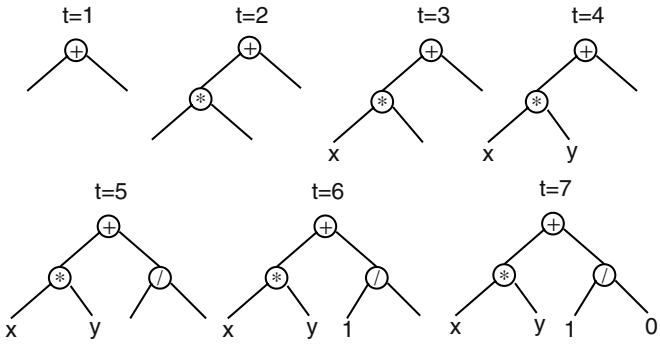

Fig. 4. Creation of a full tree having maximum depth 2 (and therefore a total of seven nodes) using the Full initialization method (t=time)

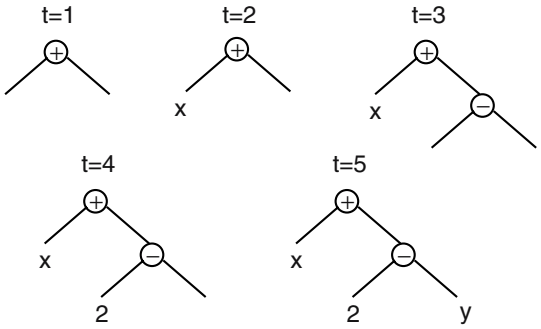

Fig. 5. Creation of a five node tree using the Grow initialization method with a maximum depth of 2 (t=time). A terminal is chosen at $t = 2$, causing the left branch of the root to be closed at that point even though the maximum depth had not been reached

must be leaves, or the resulting tree would be too deep; thus at time $t = 3$ and time $t = 4$ terminals must be chosen (`x` and `y` in this case).

Where the *Full* method generates trees of a specific size and shape, the *Grow* method allows for the creation of trees of varying size and shape. Here nodes are selected from the whole primitive set (functions *and* terminals) until the depth limit is reached, below which only terminals may be chosen (as is the case in the *Full* method). Figure 5 illustrates this process for the construction of a tree with depth limit 2. Here the first child of the root `+` node happens to be a terminal, thus closing off that branch before actually reaching the depth limit. The other child, however, is a function (`-`), but its children are forced to be terminals to ensure that the resulting tree does not exceed the depth limit.

Pseudo code for a recursive implementation of both the *Full* and *Grow* methods is given in Algorithm 2.

Algorithm 2 Pseudo code for recursive program generation with the *Full* and *Grow* methods

procedure: gen_rnd_expr(func_set, term_set, max_d, method)

```
1: if max_d = 0 or (method = grow and rand() < |term_set| / (|term_set|+|func_set|)) then
2:    expr = choose_random_element( term_set )
3: else
4:    func = choose_random_element( func_set )
5:    for i = 1 to arity(func) do
6:       arg_i = gen_rnd_expr( func_set, term_set, max_d - 1, method );
7:    expr = (func, arg_1, arg_2, ...);
8: return expr
```

Notes: **func_set** is a function set, **term_set** is a terminal set, **max_d** is the maximum allowed depth for expressions, **method** is either *Full* or *Grow* and **expr** is the generated expression in prefix notation.

Note here that the size and shapes of the trees generated via the *Grow* method are highly sensitive to the sizes of the function and terminal sets. If, for example, one has significantly more terminals than functions, the *Grow* method will almost always generate very short trees regardless of the depth limit. Similarly, if the number of functions is considerably greater than the number of terminals, then the *Grow* method will behave quite similarly to the *Full* method. While this is a particular problem for the *Grow* method, it illustrates a general issue where small (and often apparently inconsequential) changes such as the addition or removal of a few functions from the function set can in fact have significant implications for the GP system, and potentially introduce important unintended biases.

Because neither the *Grow* or *Full* method provide a very wide array of sizes or shapes on their own, Koza proposed a combination called *ramped half-and-half* [188]. Here half the initial population is constructed using *Full* and half is constructed using *Grow*. This is done using a range of depth limits (hence the term 'ramped') to help ensure that we generate trees having a variety of sizes and shapes.

While these methods are easy to implement and use, they often make it difficult to control the statistical distributions of important properties such as the sizes and shapes of the generated trees. Other initialization mechanisms, however, have been developed (such as [239]) that do allow for closer control of these properties in instances where such control is important.

It is also worth noting that the initial population need not be entirely random. If something is known about likely properties of the desired solution, trees having these properties can be used to seed the initial population. Such seeds might be created by humans based on knowledge of the problem domain, or they could be the results of previous GP runs. However, one needs to be careful not to create a situation where the second generation is dominated

by offspring of a single or very small number of seeds. Diversity preserving techniques, such as multi-objective GP [287, 344], demes [203] (see Sect. 8.6), fitness sharing [115] and the use of multiple seed trees, might be used. In any case, the diversity of the population should be monitored to ensure that there is significant mixing of different initial trees.

2.3 Selection

Like in most other EAs, genetic operators in GP are applied to individuals that are probabilistically selected based on fitness. That is, better individuals are more likely to have more child programs than inferior individuals. The most commonly employed method for selecting individuals in GP is tournament selection, followed by fitness-proportionate selection, but any standard EA selection mechanism can be used. Since selection has been described elsewhere in this book (in the Chapters on EAs), we will not provide any additional details.

2.4 Recombination and Mutation

Where GP departs significantly from other EAs is in the implementation of the operators of crossover and mutation. The most commonly used form of crossover is *subtree crossover*. Given two parents, subtree crossover randomly selects a crossover point in each parent tree. Then, it creates the offspring by replacing the sub-tree rooted at the crossover point in a copy of the first parent with a copy of the sub-tree rooted at the crossover point in the second parent, as illustrated in Fig. 6.

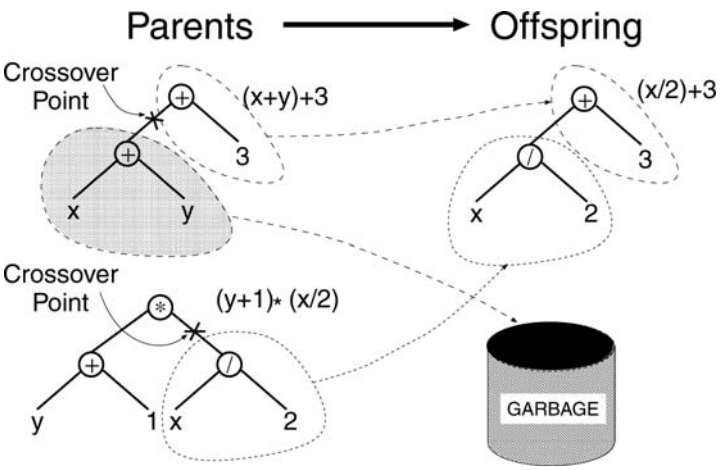

Fig. 6. Example of subtree crossover

Except in technical studies on the behavior of GP, crossover points are usually *not* selected with uniform probability. Typical GP primitive sets lead to trees with an average branching factor of at least two, so the majority of the nodes will be leaves. Consequently the uniform selection of crossover points leads to crossover operations frequently exchanging only very small amounts of genetic material (that is, small subtrees); many crossovers may in fact reduce to simply swapping two leaves. To counter this, Koza suggested the widely used approach of choosing functions 90% of the time, while leaves are selected 10% of the time.

While subtree crossover is the most common version of crossover in tree-based GP, other forms have been defined and used. For example, *one-point crossover* [222, 298, 300] works by selecting a *common* crossover point in the parent programs and then swapping the corresponding subtrees. To account for the possible structural diversity of the two parents, one-point crossover analyzes the two trees from the root nodes and considers for the selection of the crossover point only the parts of the two trees, called the *common region*, which have the same topology (that is, the same arity in the nodes encountered traversing the trees from the root node). In *context-preserving crossover* [79], the crossover points are constrained to have the same coordinates, like in one-point crossover. However, in this case no other constraint is imposed on their selection (in other words, they are not limited to the common region).

In *size-fair crossover* [205, 206] the first crossover point is selected randomly like in standard crossover. Then the size of the subtree to be excised from the first parent is calculated. This is used to constrain the choice of the second crossover point so as to guarantee that the subtree excised from the second parent will not be 'unfairly' big. Finally, it is worth mentioning that the notion of common region is related to the notion of homology, in the sense that the common region represents the result of a matching process between parent trees. It is then possible to imagine that within such a region transfer of homologous primitives can happen in very much like the same way as it happens in GAs operating on linear chromosomes. An example of recombination operator that implements this idea is *uniform crossover* for GP [299].

The most commonly used form of mutation in GP (which we will call *subtree mutation*) randomly selects a mutation point in a tree and substitutes the sub-tree rooted there with a randomly generated sub-tree. This is illustrated in Fig. 7. Subtree mutation is sometimes implemented as crossover between a program and a newly generated random program; this operation is also known as 'headless chicken' crossover [10].

Another common form of mutation is *point mutation*, which is the rough equivalent for GP of the bit-flip mutation used in GAs. In point mutation a random node is selected and the primitive stored there is replaced with a different random primitive of the same arity taken from the primitive set. If no other primitives with that arity exist, nothing happens to that node (but other

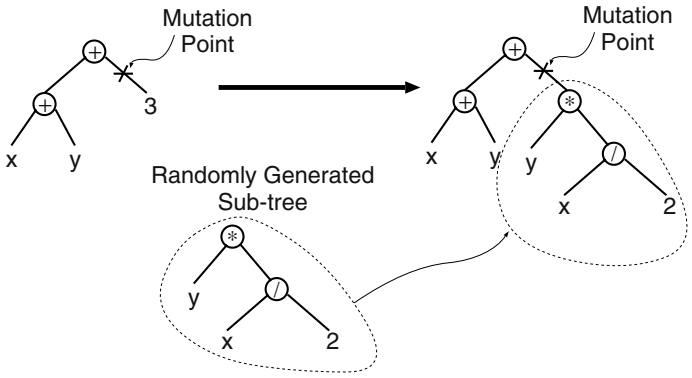

Fig. 7. Example of subtree mutation

nodes may still be mutated). Note that, when subtree mutation is applied, this involves the modification of exactly one subtree. Point mutation, on the other hand, is typically applied with a given mutation rate on a per-node basis, allowing multiple nodes to be mutated independently.

There are a number of mutation operators which treat constants in the program as special cases. [341] mutates constants by adding Gaussianly distributed random noise to them. However, others use a variety of potentially expensive optimization tools to try and fine tune an existing program by finding the 'best' value for constants within it. For instance, [340] uses 'a numerical partial gradient ascent ... to reach the nearest local optimum' to modify all constants in a program, while [348] uses simulated annealing to stochastically update numerical values within individuals.

While mutation is not necessary for GP to solve many problems, [281] argues that, in some cases, GP with mutation alone can perform as well as GP using crossover. While mutation was often used sparsely in early GP work, it is more widely used in GP today, especially in modeling applications.

3 Getting Ready to Run Genetic Programming

To run a GP system to solve a problem a small number of ingredients, often termed *preparatory steps*, need to be specified:

1. the terminal set,
2. the function set,
3. the fitness measure,
4. certain parameters for controlling the run, and
5. the termination criterion and method for designating the result of the run.

In this Section we consider these ingredients in more detail.

3.1 Step 1: Terminal Set

While it is common to describe GP as evolving *programs*, GP is not typically used to evolve programs in the familiar, Turing-complete languages humans normally use for software development. It is instead more common to evolve programs (or expressions or formulae) in a more constrained and often domain-specific language. The first two preparatory steps, the definition of the terminal and function sets, specify such a language – that is, the ingredients that are available to GP to create computer programs.

The terminal set may consist of:

- *the program's external inputs*, typically taking the form of named variables (say `x`, `y`);
- *functions with no arguments*, which are, therefore, interesting either because they return different values in different invocations (for example, the function `rand()` that returns random numbers, or a function `dist_to_wall()` that returns the distance from the robot we are controlling to an obstacle) or because the produce side effects (such as `go_left()`); and
- *constants*, which can either be pre-specified or randomly generated as part of the tree creation process.

Note that using a primitive such as `rand` can cause the behavior of an individual program to vary every time it is called, even if it is given the same inputs. What we often want instead is a set of fixed random constants that are generated as part of the process of initializing the population. This is typically accomplished by introducing a terminal that represents an *ephemeral random constant*. Every time this terminal is chosen in the construction of an initial tree (or a new subtree to use in an operation like mutation), a different random value is generated which is then used for that *particular* terminal, and which will remain fixed for the rest of the run. The use of ephemeral random constants is typically denoted by including the symbol $\Re$ in the terminal set; see Sect. 4 for an example.

3.2 Step 2: Function Set

The function set used in GP is typically driven by the nature of the problem domain. In a simple numeric problem, for example, the function set may consist of merely the arithmetic functions (`+`, `-`, `*`, `/`). However, all sorts of other functions and constructs typically encountered in computer programs can be used. Table 1 shows a sample of some of the functions one sees in the GP literature. Also for many problems, the primitive set includes specialised functions and terminals which are expressly designed to solve problems in a specific domain of application. For example, if the goal is to program a robot to mop the floor, then the function set might include such actions as `move`, `turn`, and `swish-the-mop`.

Table 1. Examples of primitives allowed in the GP function and terminal sets

Function Set	
Kind of Primitive	*Example(s)*
Arithmetic	`+`, `*`, `/`
Mathematical	`sin`, `cos`, `exp`
Boolean	`AND`, `OR`, `NOT`
Conditional	`IF-THEN-ELSE`
Looping	`FOR`, `REPEAT`
⋮	⋮

Terminal Set	
Kind of Primitive	*Example(s)*
Variables	`x`, `y`
Constant values	`3`, `0.45`
0-arity functions	`rand`, `go_left`

Closure

For GP to work effectively, most function sets are required to have an important property known as *closure* [188], which can in turn be broken down into the properties of *type consistency* and *evaluation safety*.

Type consistency is necessitated by the fact that subtree crossover (as described in Sect. 2.4) can mix and join nodes quite arbitrarily during the evolutionary process. As a result it is necessary that *any* subtree can be used in any of the argument positions for every function in the function set, because it is always possible that sub-tree crossover will generate that combination. For functions that return a value (such as `+`, `-`, `*`, `/`), it is then common to require that all the functions be type consistent, namely that they all return values of the same type, and that all their arguments be of that type as well. In some cases this requirement can be weakened somewhat by providing an automatic conversion mechanism between types – for example, converting numbers to Booleans by treating all negative values as *false*, and non-negative values as *true*. Conversion mechanisms like this can, however, introduce unexpected biases into the search process, so they should be used thoughtfully.

The requirement of type consistency can seem quite limiting, but often simple restructuring of the functions can resolve apparent problems. An `if` function, for example, would often be defined as taking three arguments: The test, the value to return if the test evaluates to *true*, and the value to return if the test evaluates to *false*. The first of these three arguments is clearly Boolean, which would suggest that `if` can't be used with numeric functions like `+`. This can easily be worked around however by providing a mechanism to automatically convert a numeric value into a Boolean as discussed above. Alternatively, one can replace the traditional `if` with a function of four (numeric) arguments a, b, c, d with the semantics 'If $a < b$ then return value c, otherwise return value d'. These are obviously just specific examples of general techniques; the details are likely to depend on the particulars of your problem domain.

An alternative to requiring type consistency is to extend the GP system to, for example, explicitly include type information, and constrain operations like crossover so they do not perform 'illegal' (from the standpoint of the type system) operations. This is discussed further in Sect. 5.4.

The other component of closure is evaluation safety, necessitated by the fact that many commonly used functions can fail in various ways. An evolved expression might, for example, divide by 0, or call `MOVE_FORWARD` when facing a wall or precipice. This is typically dealt with by appropriately modifying the standard behavior of primitives. It is common, for example, to use *protected* versions of numeric functions that can throw exceptions, such as division, logarithm, and square root. The protected version of such a function first tests for potential problems with its input(s) before executing the corresponding instruction, and if a problem is spotted some pre-fixed value is returned. Protected division (often notated with `%`), for example, checks for the case that its second argument is 0, and typically returns 1 if it is (regardless of the value of the first argument).[1] Similarly, `MOVE_AHEAD` can be modified to do nothing if a forward move is illegal for some reason or, if there are no other obstacles, the edges can simply be eliminated by making the world toroidal.

An alternative to protected functions is to trap run-time exceptions and strongly reduce the fitness of programs that generate such errors. If the likelihood of generating invalid expressions is very high, however, this method can lead to all the individuals in the population having nearly the same (very poor) fitness, leaving selection with very little discriminatory power.

One type of run-time error that is somewhat more difficult to check for is numeric overflow. If the underlying implementation system throws some sort of exception, then this can be handled either by protection or by penalizing as discussed above. If, however, the implementation language quietly ignores the overflow (for instance, the common practice of wrapping around on integer overflow), and if this behavior is seen as unacceptable, then the implementation will need to include appropriate checks to catch and handle such overflows.

Sufficiency

There is one more property that, ideally, primitives sets should have: *sufficiency*. Sufficiency requires that the primitives in the primitive set are capable of expressing the solutions to the problem, in other words that the set of all the possible recursive compositions of such primitives includes at least one

[1] The decision to return 1 here provides the GP system with a simple and reliable way to generate the constant 1, via an expression of the form `(/ x x)`. This, combined with a similar mechanism for generating 0 via `(- x x)` ensures that GP can easily construct these two important constant.

solution. Unfortunately, sufficiency can be guaranteed only for some problems, when theory or experience with other methods tells us that a solution can be obtained by combining the elements of the primitive set.

As an example of a sufficient primitive set let us consider the set {`AND`, `OR`, `NOT`, `x1`, `x2`, ..., `xN`}, which is always sufficient for Boolean function induction problems, since it can produce all Boolean functions of the variables `x1`, `x2`, ..., `xN`. An example of insufficient set is the set {`+`, `-`, `*`, `/`, `x`, `0`, `1`, `2`}, which is insufficient whenever, for example, the target function is transcendental – for example, $\exp(x)$ – and therefore cannot be expressed as a rational function (basically, a ratio of polynomials). When a primitive set is insufficient for a particular application, GP can only develop programs that approximate the desired one, although perhaps very closely.

Evolving Structures other than Programs

There are many problems in the real world where solutions cannot be directly cast as computer programs. For example, in many design problems the solution is an artifact of some type (a bridge, a circuit, or similar). GP has been applied to problems of this kind by using a trick: the primitive set is designed in such a way that, through their execution, programs construct solutions to the problem. This has been viewed as analogous to the development by which an egg grows into an adult. For example, if the goal is the automatic creation of an electronic controller for a plant, the function set might include common components such as `integrator`, `differentiator`, `lead`, `lag`, and `gain`, and the terminal set might contain `reference`, `signal`, and `plant output`. Each of these operations, when executed, then insert the corresponding device into the controller being built. If, on the other hand, the goal is the synthesis of analogue electrical circuits the function set might include components such as transistors, capacitors, resistors, and so on. This is further discussed in Sect. 5.5.

3.3 Step 3: Fitness Function

The first two preparatory steps define the primitive set for GP, and therefore, indirectly define the search space GP will explore. This includes all the programs that can be constructed by composing the primitives in all possible ways. However, at this stage we still do not know which elements or regions of this search space are good (that is, include programs that solve or approximately solve the problem). This is the task of the fitness measure, which effectively (albeit implicitly) specifies the desired goal of the search process. The fitness measure is our primary (and often sole) mechanism for giving a 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 computes its square root).

Depending on the problem at hand, fitness can be measured in terms of the amount of *error* between its output and the desired output, the amount of *time* (fuel, money, and the like) required to bring a system to a desired *target state*, the *accuracy* of the program in recognizing patterns or classifying objects into classes, the *payoff* that a game-playing program produces, the *compliance* of a structure with user-specified design criteria, and so on.

There is something unusual about the fitness functions used in GP that differentiates them from those used in most other EAs. Because the structures being evolved in GP are computer programs, fitness evaluation normally requires executing all the programs in the population, typically multiple times. While one can compile the GP programs that make up the population, the overhead is usually substantial, so it is much more common to use an interpreter to evaluate the evolved programs.

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 recursively 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. 8, where the number to the right of each internal node represents the result of evaluating the subtree root at that node. In this example, the independent variable `X` evaluates to `-1`. Algorithm 3 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.

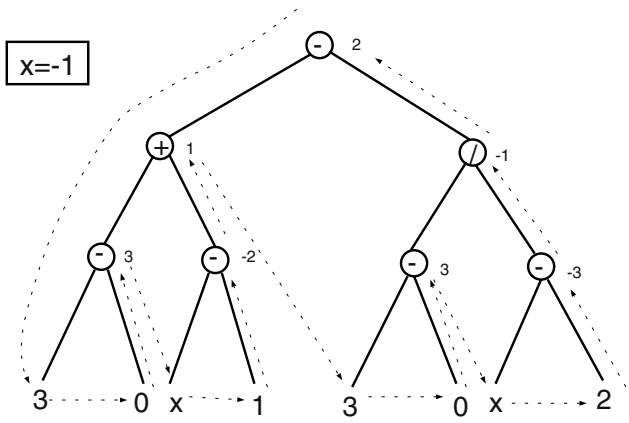

Fig. 8. Example interpretation of a syntax tree (the terminal x is a variable has a value of −1). The number to the right of each internal node represents the result of evaluating the subtree root at that node

Algorithm 3 Typical interpreter for GP

```
procedure: eval( expr )
 1: if expr is a list then
 2:    proc = expr(1) {Non-terminal: extract root}
 3:    if proc is a function then
 4:       value = proc( eval(expr(2)), eval(expr(3)), ... ) {Function: evaluate
          arguments}
 5:    else
 6:       value = proc( expr(2), expr(3), ...) {Macro: don't evaluate arguments}
 7: else
 8:    if expr is a variable or expr is a constant then
 9:       value = expr {Terminal variable or constant: just read the value}
10:    else
11:       value = expr() {Terminal 0-arity function: execute}
12: return value
```

Notes: **expr** is an expression in prefix notation, **expr(1)** represents the primitive at the root of the expression, **expr(2)** represents the first argument of that primitive, **expr(3)** represents the second argument, and so forth.

In some problems we are interested in the *output* produced by a program, namely the value returned when we evaluate starting at the root node. In other problems, however, we are interested in the actions performed by a program. In this case the primitive set will include functions with side effects – that is, functions that do more than just return a value – but say change some global data structures, print or draw something on the screen or control the motors of a robot. Irrespective of whether we are interested in program outputs or side effects, quite often the fitness of a program depends on the results produced by its execution on many different inputs or under a variety of different conditions. These different test cases typically incrementally contribute to the fitness value of a program, and for this reason are called *fitness cases*.

Another common feature of GP fitness measures is that, for many practical problems, they are *multi-objective*, in other words they combine two or more different elements that are often in competition with one another. The area of multi-objective optimization is a complex and active area of research in GP and machine learning in general; see [73], for example, for more.

3.4 Steps 4 and 5: Parameters and Termination

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. Typically the single best-so-far individual is then harvested and designated as the result of the run, although one might wish to return additional individuals and data as necessary or appropriate for your problem domain.

4 Example Genetic Programming Run

This Section provides a concrete, illustrative run of GP in which the goal is to automatically evolve an expression whose values match those of the quadratic polynomial $x^2 + x + 1$ in the range $[-1, +1]$. 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* (see Sect. 7.1 for more).

We begin with the five preparatory steps from the previous section, and then describe in detail the events in one possible run.

4.1 Preparatory Steps

The purpose of the first two preparatory steps is to specify the ingredients the evolutionary process can use to construct potential solutions. Because the problem is to find a mathematical function of one independent variable, x, the terminal set (the inputs to the to-be-evolved programs) must include this variable. The terminal set also includes ephemeral random constants, drawn from some reasonable range, say from -5.0 to $+5.0$, as described in Sect. 3.1. Thus the terminal set, T, is

$$T = \{x, \Re\} \tag{1}$$

The 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 simple choice for the function set consists of the four ordinary arithmetic functions: addition, subtraction, multiplication, and division. Most numeric regression will include at least these operations, often in conjunction with additional functions such as sin and log. In our example, however, we will restrict ourselves to the simple function set

$$F = \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{\%}\} \tag{2}$$

where % is protected division as discussed in Sect. 3.2.

The third preparatory step involves constructing the fitness measure that specifies 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 integral of the absolute value of the differences (errors) between the individual mathematical expression and the target quadratic polynomial $x^2 + x + 1$, taken over the range $[-1, +1]$. However, for most symbolic regression problems, it is not practical or possible to analytically compute the value of the integral of the absolute error. Thus it is common to instead define the fitness to be the *sum of absolute errors* measured at different values of the independent variable x in the range $[-1.0, +1.0]$. In particular, we will measure the errors for $x = -1.0, -0.9, \cdots, 0.9, 1.0$. A smaller value of fitness (error) is better; a fitness (error) of zero would indicate a perfect fit. Note that with this definition, our fitness is (approximately) proportional to the area between the parabola $x^2 + x + 1$ and the curve representing the candidate individual (see Fig. 10 for examples).

The fourth step is where we set our run parameters. The population size in this small illustrative example will be just four. In actual practice, the population size for a run of GP typically consists of thousands or millions of individuals, but we will use this tiny population size to keep the example manageable. In practice, the crossover operation is commonly used to generate about 90% of the individuals in the population; the reproduction operation (where a fit individual is simply copied from one generation to the next) is used to generate about 8% of the population; the mutation operation is used to generate about 1% of the population; and the architecture-altering operations (see Sect. 5.2) are used to generate perhaps 1% of the population. Because this example involves an abnormally small population of only four individuals, the crossover operation will be used to generate two individuals, and the mutation and reproduction operations will each be used to generate one individual. For simplicity, the architecture-altering operations are not used for this problem.

In the fifth and final step we need to specify a termination condition. A reasonable termination criterion for this problem is that the run will continue from generation to generation until the fitness (or error) of some individual is less than 0.1. In this contrived example, our example run will (atypically) yield an algebraically perfect solution (with a fitness of zero) after merely one generation.

4.2 Step-by-Step Sample Run

Now that we have performed the five preparatory steps, the run of GP can be launched.

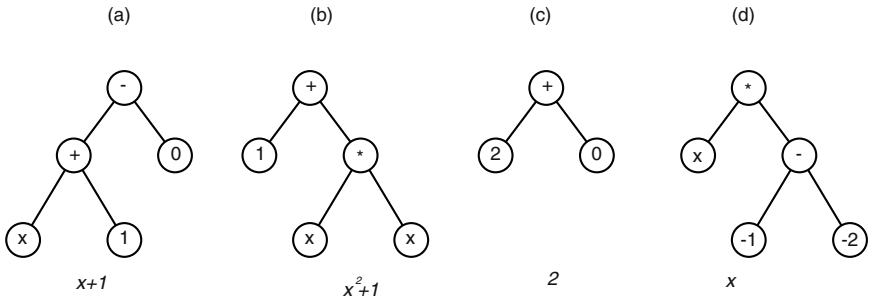

Fig. 9. Initial population of four randomly created individuals of generation 0

Initialization

GP starts by randomly creating a population of four individual computer programs. The four programs are shown in Fig. 9 in the form of trees.

The first randomly constructed program tree (Fig. 9a), and is equivalent to the expression $x+1$. The second program (Fig. 9b) adds the constant terminal 1 to the result of multiplying x by x and is equivalent to x^2+1. The third program (Fig. 9c) adds the constant terminal 2 to the constant terminal 0 and is equivalent to the constant value 2. The fourth program (Fig. 9d) is equivalent to x.

Fitness Evaluation

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. Here, for example, the four random individuals from generation 0 in Fig. 9 produce outputs that deviate by different amounts from the target function x^2+x+1. Fig. 10 compares the plots of each of the four individuals in Fig. 9 and the target quadratic function x^2+x+1. The sum of absolute errors for the straight line $x+1$ (the first individual) is 7.7 (Fig. 10a). The sum of absolute errors for the parabola x^2+1 (the second individual) is 11.0 (Fig. 10b). The sums of the absolute errors for the remaining two individuals are 17.98 (Fig. 10c) and 28.7 (Fig. 10d), respectively.

As can be seen in Fig. 10, the straight line $x+1$ (Fig. 10a) is closer to the parabola x^2+x+1 in the range from –1 to +1 than any of three other programs in the population. This straight line is, of course, not equivalent to the parabola x^2+x+1; it is not even a quadratic function. It is merely the best candidate that happened to emerge from the blind (and very limited) random search of generation 0. In the valley of the blind, the one-eyed man is king.

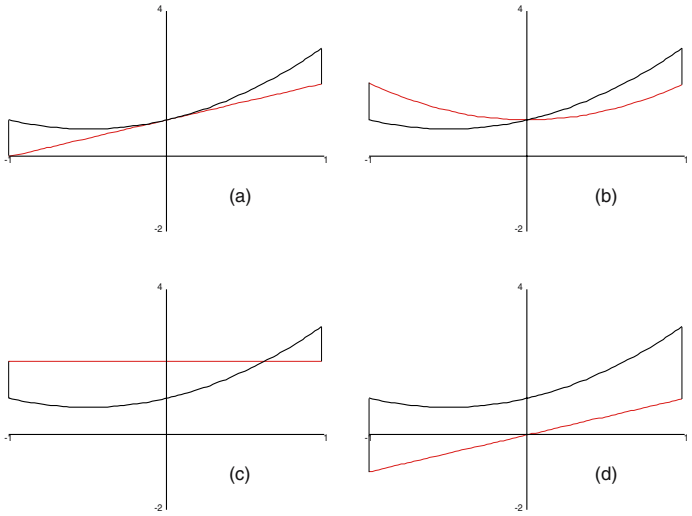

Fig. 10. Graphs of the evolved functions from generation 0. The heavy line in each plot is the target function x^2+x+1, with the other line being the evolved functions from the first generation (see Fig. 9). The fitness of each of the four randomly created individuals of generation 0 is approximately proportional to the area between two curves, with the actual fitness values being 7.7, 11.0, 17.98 and 28.7 for individuals (**a**) through (**d**), respectively

Selection, Crossover and Mutation

After the fitness of each individual in the population is ascertained, GP then probabilistically selects relatively more fit programs from the population to act as the parents of the next generation. The genetic operations are applied to the selected individuals to create offspring programs. The important point for our example is that our selection process 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 and the worst individual in the population will not necessarily be excluded.

In this example, we will start with the reproduction operation. Because the first individual (Fig. 9a) is the most fit individual in the population, it is very likely to be selected to participate in a genetic operation. Let us suppose that this particular individual is, in fact, selected for reproduction. If so, it is copied, without alteration, into the next generation (generation 1). It is shown in Fig. 11a 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. 9c) 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

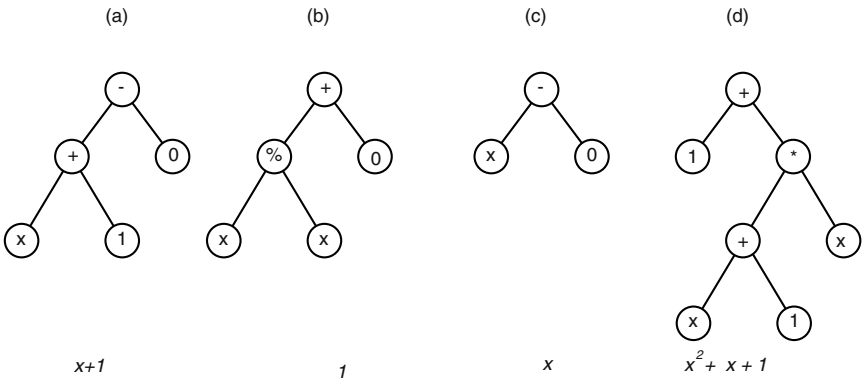

Fig. 11. Population of generation 1 (after one reproduction, one mutation, and one two-offspring crossover operation)

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 constructed 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. 11b. This particular mutation changes the original individual from one having a constant value of 2 into one having a constant value of 1, improving its fitness from 17.98 to 11.0.

Finally, we use the crossover operation to generate our final two individuals for the next generation. Because the first and second individuals in generation 0 are both relatively fit, they are likely to be selected to participate in crossover. However, selection can always pick suboptimal individuals. So, let us assume that in our first application of crossover the pair of selected parents is composed of the above-average tree in Figs. 9a and the below-average tree in Fig. 9d. One point of the first parent, namely the + function in Fig. 9a, is randomly picked as the crossover point for the first parent. One point of the second parent, namely the leftmost terminal x in Fig. 9d, is randomly picked as the crossover point for the second parent. The crossover operation is then performed on the two parents. The offspring (Fig. 11c) is equivalent to x and is not particularly noteworthy.

Let us now assume, that in our second application of crossover, selection chooses the two most fit individuals as parents: the individual in Fig. 9b as the first parent, and the individual in Fig. 9a as the second. Let us further imagine that crossover picks the leftmost terminal x in Fig. 9b as a crossover point for the first parent, and the + function in Fig. 9a as the crossover point for the second parent. Now the offspring (Fig. 11d) is equivalent to $x^2 + x + 1$ and has a fitness (sum of absolute errors) of zero. Because the fitness of this

individual is below 0.1, the termination criterion for the run is satisfied and the run is automatically terminated. This best-so-far individual (Fig. 11d) is then designated as the result of the run.

Note that the best-of-run individual (Fig. 11d) incorporates a good trait (the quadratic term x^2) from the first parent (Fig. 9b) with two other good traits (the linear term x and constant term of 1) from the second parent (Fig. 9a). The crossover operation thus produced a solution to this problem by recombining good traits from these two relatively fit parents into a superior (indeed, perfect) offspring.

This is, obviously, a highly simplified example, and the dynamics of a real GP run are typically far more complex than what is presented here. Also, in general there is no guarantee that an exact solution like this will be found by GP.

5 Advanced Tree-Based GP Techniques

5.1 Automatically Defined Functions

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 makes it possible to exploit a problem's modularities, symmetries, and regularities (and thereby potentially accelerate 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, and so forth.

While several different mechanisms for evolving reusable components have been proposed (for instance, [13, 331]), Koza's *Automatically Defined Functions* (ADFs) [189] have been the most successful way of evolving reusable components.

When ADFs are used, a program consists of one (or more) function-defining trees (that is, ADFs) as well as one or more main result-producing trees (see Fig. 3). An ADF may have none, one, or more inputs. The body of an ADF contains its work-performing steps. Each ADF belongs to a particular program in the population. An ADF may be called by the program's main result-producing tree, by another ADF, or by another type of tree (such as the other types of automatically evolved program components described below). Recursion is sometimes allowed. Typically, the ADFs are called with different inputs. The work-performing steps of the program's main result-producing tree and the work-performing steps of each ADF are automatically and simultaneously created by GP. The program's main result-producing tree and its

ADFs typically have different function and terminal sets. ADFs are the focus of [189] and [190].

Koza also proposed other types of automatically evolved program components. Automatically defined iterations (ADIs), automatically defined loops (ADLs) and automatically defined recursions (ADRs) provide means (in addition to ADFs) to reuse code. Automatically defined stores (ADSs) provide means to reuse the result of executing code. These automatically defined components are described in [195].

5.2 Program Architecture and Architecture-Altering Operations

The architecture of a program consists of the total number of trees, the type of each tree (for example, result-producing tree, ADF, ADI, ADL, ADR, or ADS), 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 tree.

There are three ways to determine the architecture of the computer programs that will be evolved:

1. The human user may specify in advance the architecture of the overall program, in other words perform an architecture-defining preparatory step in addition to the five itemized in Sect. 2.
2. The run may employ evolutionary selection of the architecture (as described in [189]), thereby enabling the architecture of the overall program to emerge from a competitive process during the run of GP.
3. The run may employ a set of *architecture-altering operations* which can create new ADFs, remove ADFs, and increase or decrease the number of inputs an ADF has. Note initially, many architecture changes (such as those define in [189]) are designed not to change the semantics of the program and, so, the altered program often has exactly the same fitness as its parent. However, the new arrangement of ADFs may make it easier for subsequent changes to evolve better programs later.

5.3 Genetic Programming Problem Solver

The Genetic Programming Problem Solver (GPPS) is described in part 4 of [195]. It is a very powerful AI approach, but typically it requires considerable computational time.

When GPPS is used, the user does not need to chose either the terminal set or the function set (the first and second preparatory steps – Sect. 2). The function set for GPPS is the four basic arithmetic functions (addition, subtraction, multiplication, and division) and a conditional operator `IF`. The terminal set for GPPS consists of numerical constants and a set of input terminals that are presented in the form of a vector. By employing this generic

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

GPPS relies on the architecture-altering operations described in Sect. 5.2 to dynamically create, duplicate, and delete subroutines and loops during the run of GP. Additionally, in version 2.0 of GPPS [195, Chapter 22], the architecture-altering operations are used to dynamically create, duplicate, and delete recursions and internal storage. Because the architecture of the evolving program is automatically determined during the run, GPPS eliminates the need for the user to specify in advance whether to employ subroutines, loops, recursions and internal storage in solving a given problem. It similarly eliminates the need for the user to specify the number of arguments possessed by each subroutine. Further, GPPS eliminates the need for the user to specify the hierarchical arrangement of the invocations of the subroutines, loops, and recursions.

5.4 Constraining Syntactic Structures

As discussed in Sect. 3, most GP systems require type consistency where all sub-trees return data of the same type, ensuring that the output of any subtree can be used as one of the inputs to any other node. This ensures that the shuffling caused by sub-tree crossover, and so on, doesn't lead to incompatible connections between nodes. Many problem domains, however, have multiple types and do not naturally satisfy the type consistency requirement. This can often be addressed through creative definitions of functions and implicit type conversion, but this may not always be desirable. For example, if a key goal is that the evolved solutions should be easily understood or analyzed, then removing type concepts and other common constraints may lead to solutions that are unacceptable because they are quite difficult to interpret. GP systems that are constrained structurally or via a type system often generate results that are easier for humans to understand and analyze [137], [203, p. 126].

In this Section we will look at three different approaches to constraining the syntax of the evolved expression trees in GP: simple structure enforcement, strongly typed GP and grammar-based constraints.

Enforcing Particular Structures

If a particular structure is believed or known to be important then one can modify the GP system to require all individuals to have that structure [188]. A periodic function, for example, might be believed to be of the form $a\sin(bt)$ and so the GP is restricted to evolving expressions having that structure. (in other words, a and b are allowed to evolve freely, but the rest of the structure is fixed). Syntax restriction can also be used to make GP follow sensible engineering practices. For example, we might want to ensure that loop control variables appear in the correct parts of `FOR` loops and nowhere else [203, p.126].

This can be implemented in a number of ways. One could, for example, ensure that all the initial individuals have that structure (for example, generating random sub-trees for a and b while fixing the rest), and then constrain operations like crossover so that they do not alter any of the fixed regions. An alternative approach would be to evolve the various (sub)components separately in any of several ways. One could, for example, evolve pairs of trees (a, b), or one could have two separate populations, one of which is being used to evolve candidates for a while the other is evolving candidates for b.

Strongly Typed GP

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 [259]. 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 then constrained to not violate the type system's constraints.

Returning to the `if` example from Sect. 3, we might have a domain with both numeric and Boolean terminals (for instance, `get_speed` and `is_food_ahead`). We might then have an `if` function that takes three arguments: A test (Boolean), the value to return if the test is *true*, and the value to return if the test is *false*. Assuming that the second and third values are constrained to be numeric, then the output of the `if` is also going to be numeric. If we choose the test argument as a root parent crossover point, then the sub-tree to insert must have a Boolean output; if we choose either the second or third argument as a root parent crossover point, then the inserted sub-tree must be numeric.

This basic approach to types can be extended to more complex type systems including simple generics [259], multi-level type systems [138], and fully polymorphic, higher-order type systems with generics [413].

Grammar-Based Constraints

Another natural way to express constraints is via grammars, and these have been used in GP in a variety of ways [123,143,279,395,404]. Many of these simply use a grammar as a means of expressing the kinds of constraints discussed above in Sect. 5.4. One could enforce the structure for the period function using a grammar such as the following:

$$\begin{aligned} \mathit{tree} &::= E \times \sin(E \times t) \\ E &::= \mathit{var} \;\mid\; E\ \texttt{op}\ E \\ \texttt{op} &::= + \;\mid\; - \;\mid\; \times \;\mid\; \div \\ \texttt{var} &::= x \;\mid\; y \;\mid\; z \end{aligned} \tag{3}$$

Genetic operators are restricted to only swapping sub-trees deriving from a common non-terminal symbol in the grammar. So, for example, an E could be replaced by another E, but an E could not be replaced by an `op`. This can be extended to, for example, context-sensitive grammars by incorporating various related concepts from computational linguistics. The TAG3P+ system [143,144], for example, uses tree-adjoining grammars (TAGs) to constrain the evolved structures.

Another major area is grammatical evolution (GE) [279, 336]. In GE a grammar is used as in the example above. However instead of representing individuals directly using either expression or derivation trees, grammatical evolution represents individuals using a variable length sequence of integers. For each production rule, the set of options on the right hand side are numbered from 0 upwards. In the example above the first rule only has one option on the right hand side; this would both be numbered 0. The second rule has two options, which would be numbered 0 and 1, the third rule has four options which would be numbered 0 to 3, and the fourth rule has three options numbered 0 to 2. An expression tree is then generated by using the values in the individual to 'choose' which option to take in the production rules, rewriting the left-most non-terminal is the current expression.

If, for example, an individual is represented by the sequence:

$$39, 7, 2, 83, 66, 92, 57, 80, 47, 94$$

then the translation process would proceed as follows (with the non-terminal to be rewritten underlined in each case):

$\underline{tree}$
$\rightarrow$ $\langle$ 39 mod 1 = 0, that is, there is only one option $\rangle$
$\underline{E} \times \sin(E \times t)$
$\rightarrow$ $\langle$ 7 mod 2 = 1, in other words, choose second option $\rangle$
$(\underline{E}\ \texttt{op}\ E) \times \sin(E \times t)$
$\rightarrow$ $\langle$ 2 mod 2 = 0, namely, take the first option $\rangle$
$(\underline{\texttt{const}}\ \texttt{op}\ E) \times \sin(E \times t)$
$\rightarrow$ $\langle$ 83 mod 3 = 2, again, only one option, generate an ephemeral constant $\rangle$
$(z\ \underline{\texttt{op}}\ E) \times \sin(E \times t)$
$\rightarrow$ $\langle$ 66 mod 4 = 2, take the third option $\rangle$
$(z\ \times\ \underline{E}) \times \sin(E \times t)$
...
$(z\ \times\ x) \times \sin(z \times t)$

In this example we didn't need to use all the numbers in the sequence to generate a complete expression free of non-terminals; 94 was in fact never used. In general 'extra' genetic material is simply ignored. Alternatively, sometimes a sequence can be 'too short' in the sense that the end of the sequence

is reached before the translation process is complete. There are a variety of options in this case, including failure (assigning this individual the worst possible fitness) and wrapping (continuing the translation process, moving back to the front of the numeric sequence). See [279] for further details on this and other aspects of grammatical evolution.

Constraints and Bias

While increasing the expressive power of a type system or other constraint mechanism may indeed limit the search space by restricting the kinds of structures that can be constructed, this often comes at a price. An expressive type system typically requires more complex machinery to support it. It often makes it more difficult to generate type-correct individuals in the initial population, and more difficult to find genetic operations that do not violate the type system. In an extreme case like constructive type theory, the type system is so powerful that it can completely express the formal specification of the program, so any program/expression having this type is guaranteed to meet that specification. In the GP context this would mean that all the members of the initial population (assuming that they are required to have the desired type) would in fact be solutions to the problem, thus removing the need for any evolution at all! Even without such extreme constraints, it has often been found necessary to develop additional machinery in order to efficiently generate an initial population that satisfies the necessary constraints [259, 318, 339, 413].

As a rule, systems that focus on *syntactic* constraints (such as grammar based systems) require less machinery than those that focus on *semantic* constraints (such as type systems), since it's typically easier to satisfy the syntactic constraints in a mechanistic fashion. Grammar based systems such as GE and TAG, for example, are typically simple to initialize, and require few if any constraints to be honored by the mutation and recombination operators. The work (and the bias) in these systems is much more in the design of the grammar; once that is in place there is often little additional work required of either the practitioner or the GP system to enforce the constraints implied by the grammar.

While a constraint system may limit the search space in valuable ways [318] and can improve performance on interesting problems [144], there is no general guarantee that constraint systems will make the evolutionary search process easier. There is no broad assurance, for example, that constraint systems will significantly increase the density of solutions or (perhaps more importantly) approximate solutions. While there are cases where constraint systems smooth the search landscape [144], it is also possible for constraint systems to make the search landscape more rugged by preventing genetic operations from creating intermediate forms on potentially valuable evolutionary paths. It might be useful to extend solution density studies such as those summarised in [222] to

the landscapes generated by constraint systems in order to better understand the impact of these constraints on the underlying search spaces.

In summary, while types, grammars, and other constraint systems can be powerful tools, all such systems carry biases. One, therefore, needs to be careful to explore the biases introduced by the constraints and not simply assume that they are beneficial to the search process.

5.5 Developmental Genetic Programming

When appropriate terminals, functions and/or interpreters are defined, standard GP can go beyond the production of computer programs. For example, in a technique called *cellular encoding*, programs are interpreted as sequences of instructions which modify (grow) a simple initial structure (embryo). Once the program terminates, the quality of the resulting structure is taken to be the fitness of the program. Naturally, the primitives of the language must be appropriate to grow structures in the domain of interest. 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 [121, 122, 124] and electronic circuits [193–195], as well as in numerous other domains.

One of the advantages of indirect representations such as cellular encoding is that the standard GP operators can be used to manipulate structures (such as circuits) which may have nothing in common with standard GP trees. A disadvantage is that they require an additional genotype-to-phenotype decoding step. However, when the fitness function involves complex calculations with many fitness cases the relative cost of the decoding step is often small.

5.6 Strongly Typed Autoconstructive GP – PushGP

In some ways Spector's PushGP [183, 328, 357, 364] is also a move away from constraining evolution. Push is a strongly typed tree based language which does not enforce syntactic constraints. Essentially PushGP uses evolution (i.e. genetic programming) to automatically create programs written in the Push programming language. Each of Push's types has its own stack. In addition to stacks for integers, floats, Booleans and so on, there is a stack for objects of type program. Using this code stack, Push naturally supports both recursion and program modules (see Sect. 5.1) without human pre-specification. The code stack allows an evolved program to push itself or fragments of itself onto the stack for subsequent manipulation.

Somewhat like 'core wars', PushGP can use the code stack and other operations to allow programs to construct their own crossover and other genetic operations and create their own offspring. (Spector prevents programs from simply duplicating themselves to prevent catastrophic loss of population diversity.)

6 Linear and Graph-Based GP

Until now we have been talking about the evolution of programs expressed as one or more trees which are evaluated by a suitable interpreter. This is the original and most widespread type of GP, but there are other types of GP where programs are represented in different ways. This Section will look at linear programs and graph-like (parallel) programs.

6.1 Linear Genetic Programming

There are two different reasons for trying linear GP. Basic computer architectures are fundamentally the same now as they were twenty years ago, when GP began. Almost all architectures represent computer programs in a linear fashion (albeit with control structures, jumps and loops). So, why not evolve linear programs [24, 280, 288]. Also, computers do not naturally run tree-shaped programs. So, slow interpreters have to be used as part of tree-based GP. On the contrary, by evolving the binary bit patterns actually obeyed by the computer, the use of an expensive interpreter (or compiler) is avoided and GP can run several orders of magnitude faster [65, 88, 272, 275].

The typical crossover and mutation operators for linear GP ignore the details of the machine code of the computer being used. For example, crossover typically chooses randomly two crossover points in each parent and swaps the code lying between them. Since the crossed over fragments are typically of different lengths, such a crossover may change the programs' lengths (see Fig. 12). Since computer machine code is organised into 32- or 64-bit words, the crossover points occur only at the boundaries between words. Therefore, a whole number of words, containing a whole number of instructions are typically swapped over. Similarly, mutation operations normally respect word boundaries and generate legal machine code. However, linear GP lends itself to a variety of other genetic operations. For example, Fig. 13 shows homologous crossover. Many other crossover and mutation operations are possible [215].

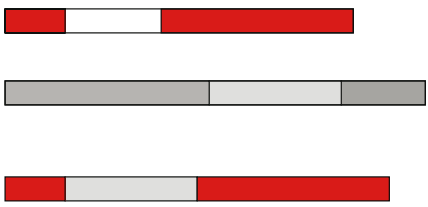

Fig. 12. Typical linear GP crossover. Two instructions are randomly chosen in each parent (*top two genomes*) as cut points. If the code fragment excised from the first parent is replaced with the code fragment excised form the second to give the child (*lower chromosome*)

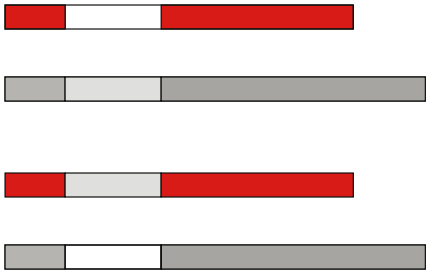

Fig. 13. Discipulus' 'homologous' crossover [99,101,275]. Two parents (*top two programs*) crossover to yield two child programs (*bottom*). The two crossover cut points are the same in both parents. Note code does not change its position relative to the start of the program (*left edge*) and the child programs are the same lengths as their parents. Homologous crossover is often combined with a small amount of normal two point crossover (Fig. 12) to introduce length changes into the GP population

If the goal is execution speed, then the evolved code should be machine code for a real computer rather than some higher level language or virtual-machine code. For example, [272] started by evolving machine code for SUN computers; [65] targeted the Z80. The linear GP of [226] was firmly targeted at novel hardware but much of the GP development had to be run in simulation whilst the hardware itself was under development.

The Sun SPARC has a simple 32-bit RISC architecture which eases designing genetic operation which manipulate its machine code. Nordin wrapped each machine code GP individual inside a C function [273]. Each of the GP program's inputs were copied from one of the C function's arguments into one of the machine registers. Note that typically there are only a small number of inputs. Linear GP should be set up to write-protect these registers, so that inputs cannot be overwritten, since if an input is overwritten and its value is lost, the evolved code cannot be a function of it. As well as the registers used for inputs, a small number (say 2–4) of other registers are used for scratch memory to store partial results of intermediate calculations. Finally, the GP simply leaves its answer in one of the registers. The external framework uses this as the C function's *return* value.

Note that execution speed is not the only reason for using linear GP. Linear programs can be interpreted, just as trees can be. Indeed a linear interpreter can be readily implemented. A simple linear structure lends itself to rapid analysis, which can be used for 'dead code' removal [33]. In some ways the search space of linear GP is easier to analyse than that of trees [204, 207–209, 215]. For example, we have used the T7 and T8 architectures (in simulation) for several large scale experimental and mathematical analysis of Turing complete GP [214,220,223,224]. For these reasons, it makes sense to consider linear 'machine' code GP, for example, in Java. Since Java is usually

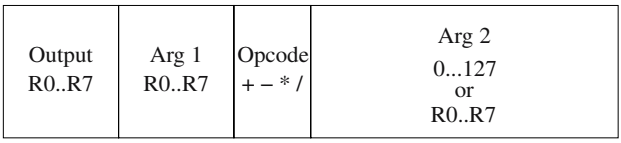

Fig. 14. Format of a linear GP engine instruction. To avoid the overhead of packing and unpacking data in the interpreter (written in a high level language such as C++), virtual machine instructions, unlike real machine instructions, are not packed into bit fields. In linear GP, instructions are laid from the start of the program to its end. In machine code GP, these are real machine code instructions. In interpreted linear GP, machine code is replaced with virtual machine code

run on a virtual machine, almost by definition this requires a virtual machine (like [226]) to interpret the evolved byte code [133, 240].

Since Unix was ported onto the x86, Intel's complex instruction set has had almost complete dominance. Seeing this, Nordin ported his Sun RISC linear GP onto Intel's CISC. Various changes were made to the genetic operations which ensure that the initial random programs are made only of legal Intel machine code and that mutation operations, which act inside the x86's 32-bit word, respect the x86's complex sub-fields. Since the x86 has instructions of different lengths, special care was taken when altering them. Typically several short instructions are packed into the 4-byte words. If there are any bytes left over, they are filled with no-operation codes. In this way best use is made of the available space, without instructions crossing 32-bit boundaries. Nordin's work led to `Discipulus` [99], which has been used from applications ranging from Bioinformatics [390] to robotics [219] and bomb disposal [78]. Generally, in linear GP instructions take the form shown in Fig. 14.

6.2 Graph-Based Genetic Programming

Trees are special types of graphs. So, it is natural to ask what would happen if one extended GP so as to be able to evolve graph-like programs. Starting from the mid 1990s researchers have proposed several extensions of GP that do just that, albeit in different ways.

For example, Poli proposed Parallel Distributed GP (PDGP) [290, 292]. PDGP is a form of GP which is suitable for the evolution of efficient highly parallel programs which effectively reuse partial results. Programs are represented in PDGP as graphs with nodes representing functions and terminals. Edges represent the flow of control and results. In the simplest form of PDGP edges are directed and unlabeled, in which case PDGP can be considered a generalization of standard GP. However, more complex representations can be used, which allow the exploration of a large space of possible programs including standard tree-like programs, logic networks, neural networks, recurrent transition networks and finite state automata. In PDGP, programs are

manipulated by special crossover and mutation operators which guarantee the syntactic correctness of the offspring. For this reason PDGP search is very efficient. PDGP programs can be executed in different ways, depending on whether nodes with side effects are used or not.

In a system called PADO (Parallel Algorithm Discovery and Orchestration), Teller and Veloso used a combination of GP and linear discrimination to obtain parallel classification programs for signals and images [375] The programs in PADO are represented as graphs, although their semantics and execution strategy are very different from those of PDGP.

In Miller's Cartesian GP [256, 257], programs are represented by linear chromosomes containing integers. These are divided into groups of three or four. Each group is associated to a position in a 2-D array. An element of the group prescribes which primitive is stored at that location in the array, while the remaining elements indicate from which other locations the inputs for that primitive should be read. So, the chromosome represents a graph-like program, which is very similar to PDGP. The main difference between the two systems is that Cartesian GP operators (mainly mutation) act at the level of the linear chromosome, while in PDGP they act directly on the graph.

It is also possible to use non-graph-based GP to evolve parallel programs. For example, Bennett used a parallel virtual machine in which several standard tree-like programs (called 'agents') would have their nodes executed in parallel with a two stage mechanism simulating parallelism of sensing actions and simple conflict resolution (prioritization) for actions with side effects [28]. [8] used GP to discover rules for cellular automata, a highly parallel computational architecture, which could solve large majority classification problems. In conjunction with an interpreter implementing a parallel virtual machine, GP can also be used to translate sequential programs into parallel ones [392], or to develop parallel programs.

7 Applications

Since its early beginnings, GP has produced a cornucopia of results. The literature, which covers more than 5000 recorded uses of GP, reports an enormous number of applications where GP has been successfully used as an automatic programming tool, a machine learner or an automatic problem-solving machine. It is impossible to list all such applications here. In the following Sections we mention a representative subset for each of the main application areas of GP (Sects. 7.1–7.10), devoting particular attention to the important areas of symbolic regression (Sect. 7.1) and human-competitive results (Sect. 7.2). We conclude the section with guidelines for the choice of application areas (Sect. 7.11).

7.1 Curve Fitting, Data Modeling, and Symbolic Regression

In principle, the possible applications of GP are as many as the applications for programs (virtually infinite). However, before one can try to solve a new problem with GP, one needs to define an appropriate fitness function. In problems where only the *side effects* of the program are of interest, the fitness function usually compares the effects of the execution of a program in some suitable environments with a desired behavior, often in a very application-dependent manner. In many problems, however, the goal is *finding a function* whose output has some desired property – for example, it matches some target values (as in the example given in Sect. 4), or it is optimum against some other criteria. This type of problem is generally known as a *symbolic regression problem.*

Many people are familiar with the notion of *regression*, which is a technique used to interpret experimental data. It consists in finding the *coefficients* of a *predefined function* such that the function best fits the data. A problem with regression analysis is that, if the fit is not good, the experimenter has to keep trying different functions until a good model for the data is found. Also, in many domains thestrong tradition of only using linear or quadratic models, even though it is possible that the data would be better fit by some other model. The problem of *symbolic regression*, instead, consists in finding a *general function* (with its coefficients) that fits the given data points. Since GP does not assume *a priori* a particular structure for the resulting function, it is well suited to this sort of discovery task. Symbolic regression was one of the earliest applications of GP [188], and continues to be a widely studied domain [45, 126, 167, 227].

The steps necessary to solve symbolic regression problems include the five preparatory steps mentioned in Sect. 2. However, while in the example in Sect. 4 the data points were computed using a simple formula, in most realistic situations the collection of an appropriate set of data points is an important and sometimes complex task. Often, for example, each point represents the (measured) values taken by some variables at a certain time in some dynamic process or in a certain repetition of an experiment.

Consider, for example, the case of using GP to evolve a *soft sensor* [161]. The intent is to evolve a function that will provide a reasonable estimate of what a sensor (in, say, a production facility) *would* report, based on data from other actual sensors in the system. This is typically done in cases where placing an actual sensor in that location would be difficult or expensive for some reason. It is necessary, however, to place at least one instance of such a sensor in a working system in order to collect the data needed to train and test the GP system. Once such a sensor is placed, one would collect the values reported by that sensor, and by all the other hard sensors that are available to the evolved function, at various times, presumably covering the various conditions the evolved system will be expected to act under.

Such experimental data typically come in large tables where numerous quantities are reported. In many case which quantity is the dependent variable, that is the thing that we want to predict (for example, the soft sensor value), and which other quantities are the independent variables – in other words, the information we want to use to make the prediction (say the hard sensor values), is pre-determined. If it is not, then the experimenter needs to make this decision before GP can be applied. Finally, in some practical situations, the data tables include hundreds or even thousands of variables. It is well-known, that in these cases the efficiency and effectiveness of any machine learning or program induction method, including GP, can dramatically drop as most of the variables are typically redundant or irrelevant, forcing the system to focus considerable energy on isolating the key features. It is then necessary to perform some form of feature selection – that is, we need to decide which independent variables to keep and which to leave out.

There are problems where more than one output (prediction) is required. For example, Table 2 shows a data set with four independent variables (left) and six dependent variables (right). The data were collected for the purpose of solving an inverse kinematics problem in the `Elvis` robot [219] (the robot is shown in Fig. 15 during the acquisition of a data sample). In situations like this, one can use GP individuals including multiple trees (as in Fig. 3), graph-based GP with multiple output nodes (see Sect. 6.2), linear GP with multiple

Table 2. Samples showing apparent size and location to both of `Elvis`' eyes of his finger tip, given various right arm actuator set points (4 degrees of freedom) – see Fig. 15. When the data are used for training, GP is asked to invert the mapping and evolve functions from data collected by both cameras showing a target location to instructions to give to `Elvis`' four arm motors so that his arm moves to the target

Arm actuator				Left eye			Right eye		
				x	y	size	x	y	size
−376	−626	1000	−360	44	10	29	−9	12	25
−372	−622	1000	−380	43	7	29	−9	12	29
−377	−627	899	−359	43	9	33	−20	14	26
−385	−635	799	−319	38	16	27	−17	22	30
−393	−643	699	−279	36	24	26	−21	25	20
−401	−651	599	−239	32	32	25	−26	28	18
−409	−659	500	−200	32	35	24	−27	31	19
−417	−667	399	−159	31	41	17	−28	36	13
−425	−675	299	−119	30	45	25	−27	39	8
−433	−683	199	−79	31	47	20	−27	43	9
−441	−691	99	−39	31	49	16	−26	45	13
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

continues for a total of 691 lines

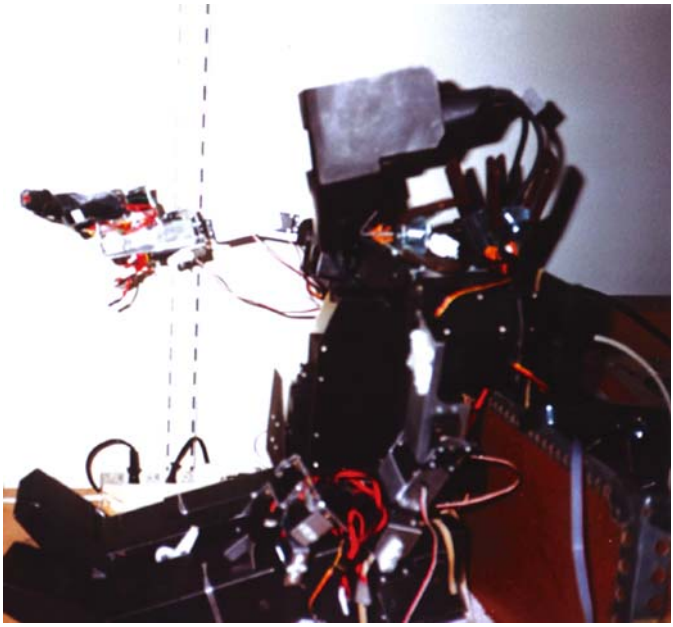

Fig. 15. `Elvis` sitting with right hand outstretched. The apparent position and size of the bright red laser attached to his finger tip is recorded (see Table 2). The data are then used to train a GP to move the robot's arm to a spot in three dimensions using only its eyes

output registers (see Sect. 6.1), a single GP tree with primitives operating on vectors, and so on.

Once a suitable data set is available, its dependent variables must all be represented in the primitive set. What other terminals and functions this will include depends very much on the type of the data being processed (are they numeric? strings?) and is often guided by information available to the experimenter on the process that generated the data. If something is known (or strongly suspected) about the desired structure of the evolved function (for example, the data is known to be periodic, so the function should probably be based on a something like sine), then applying some sort of constraint, like those discussed in Sect. 5.4, may be beneficial.

What is common to virtually all symbolic regression problems is that the fitness function must measure the ability of each program to predict the value of the dependent variable given the values of the independent ones (for each data point). So, most symbolic regression fitness functions tend to include sums over the (usually absolute or squared) errors measured for each record in the data set, as we did in Sect. 4.2.

The fourth preparatory step typically involves choosing a size for the population (which is often done initially based on the perceived difficulty of the problem, and is then refined based on the actual results of preliminary runs) and the balance between the selection strength (normally tuned via the tournament size) and the intensity of variation (which can be varied by varying the mutation and crossover rates, but many researchers tend to keep these fixed to some standard values).

7.2 Human Competitive Results – *The Humies*

Getting machines to produce human-like results is *the* reason for the existence of the fields of artificial intelligence and machine learning. However, it has always been very difficult to assess how much progress these fields have made towards their ultimate goal. Alan Turing understood that, to avoid human biases when assessing machines' intelligence, there is a need to evaluate their behavior objectively. This led him to propose an imitation game, now known as the Turing test [385]. Unfortunately, the Turing test is not usable in practice, and so, there is a need for more workable objective tests of machine intelligence.

Koza recently proposed to shift the attention from the notion of intelligence to the notion of *human competitiveness* [196]. 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. A result produced by an automated method must earn the rating of 'human competitive' independently of the fact that it was generated by an automated method.

Koza proposed that an automatically-created result should be considered 'human-competitive' if it satisfies at least one of these eight criteria:

1. 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.
2. The result is equal to or better than a result that was accepted as a new scientific result at the time when it was published in a peer-reviewed scientific journal.
3. 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.
4. The result is publishable in its own right as a new scientific result, independent of the fact that the result was mechanically created.
5. 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.
6. 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.

7. The result solves a problem of indisputable difficulty in its field.
8. 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).

These criteria are independent of and at arms-length from the fields of artificial intelligence, machine learning, and GP.

Over the years, tens of results have passed the human-competitiveness test. Some pre-2004 human-competitive results include (see [192] for a complete list):

- Creation of quantum algorithms including: a better-than-classical algorithm for a database search problem and a solution to an AND/OR query problem [361, 362].
- Creation of a competitive soccer-playing program for the `RoboCup` 1997 competition [238].
- Creation of algorithms for the transmembrane segment identification problem for proteins [189, Sects. 18.8 and 18.10] and [195, Sects. 16.5 and 17.2].
- Creation of a sorting network for seven items using only 16 steps [195, Sects. 21.4.4, 23.6, and 57.8.1].
- Synthesis of analogue circuits (with placement and routing, in some cases), including: 60- and 96-decibel amplifiers [195, Sect. 45.3]; circuits for squaring, cubing, square root, cube root, logarithm, and Gaussian functions [195, Sect. 47.5.3]; a circuit for time-optimal control of a robot [195, Sect. 48.3]; an electronic thermometer [195, Sect. 49.3]; a voltage-current conversion circuit [197, Sect. 15.4.4].
- Creation of a cellular automata rule for the majority classification problem that is better than all known rules written by humans [8].
- Synthesis of topology for controllers, including: a PID (proportional, integrative, and derivative) [197, Sect. 9.2] and a PID-D2 (proportional, integrative, derivative, and second derivative) [197, Sect. 3.7] controllers; PID tuning rules that outperform the Ziegler-Nichols and Astrom-Hagglund tuning rules [197, Chapter 12]; three non-PID controllers that outperform a PID controller that uses the Ziegler-Nichols or Astrom-Hagglund tuning rules [197, Chapter 13].

In total [192] lists 36 human-competitive results. These include 23 cases where GP has duplicated the functionality of a previously patented invention, infringed a previously patented invention, or created a patentable new invention. Specifically, there are fifteen examples where GP has created an entity that either infringes or duplicates the functionality of a previously patented 20^{th} Century invention, six instances where GP has done the same with respect to an invention patented after January 1, 2000, and two cases where GP has created a patentable new invention. The two new inventions are

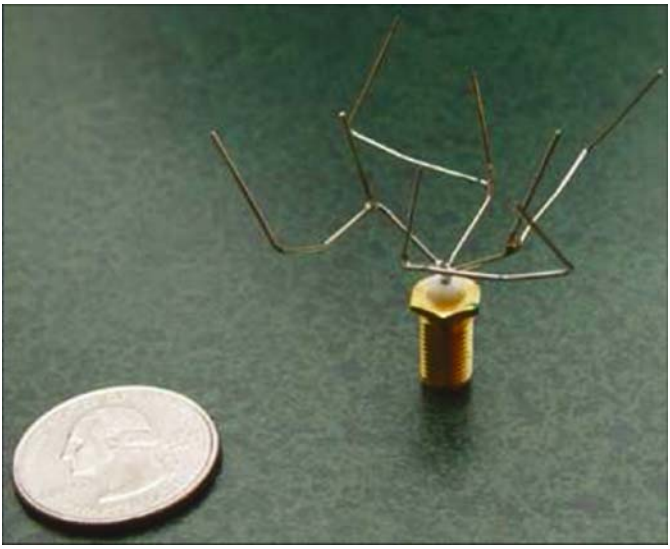

Fig. 16. Award winning human-competitive antenna design produced by GP

general-purpose controllers that outperform controllers employing tuning rules that have been in widespread use in industry for most of the 20th Century.

Many of the pre-2004 results were obtained by Koza. However, since 2004, a competition is held annually at ACM's *Genetic and Evolutionary Computation Conference* (termed the 'Human-Competitive awards – the 'Humies'). The $10,000 prize is awarded to applications that have produced automatically-created results which are equivalent to human achievements or, better.

The Humies Prizes have typically been awarded to applications of EC to high-tech fields. Many used GP. For example, the 2004 gold medals were given for the design, via GP, of an antenna for deployment on NASA's Space Technology 5 Mission (see Fig. 16) [233] and for evolutionary quantum computer programming [358]. There were three silver medals in 2004: one for evolving local search heuristics for SAT using GP [104], one for the application of GP to the synthesis of complex kinematic mechanisms [231], and one for organization design optimization using GP [175, 176]. Also, four of the 2005 medals were awarded for GP applications: the invention of optical lens systems [1, 198], the evolution of quantum Fourier transform algorithm [247], evolving assembly programs for `Core War` [61], and various high-performance game players for Backgammon, Robocode and Chess endgame [16, 17, 135, 350]. In 2006 GP again scored a gold medal with the synthesis of interest point detectors for image analysis [381, 382], while it scored a silver medal in 2007 with

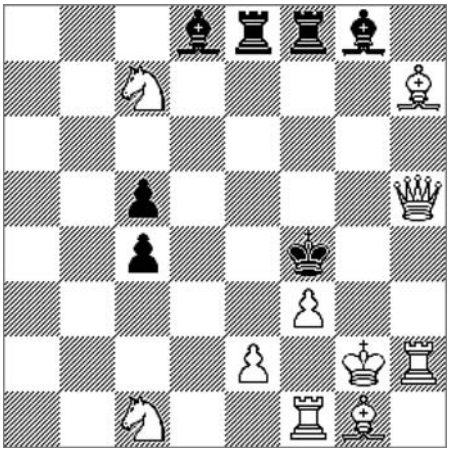

Fig. 17. Example mate-in-2 problem

the evolution of an efficient search algorithm for the Mate-in-N problem in Chess [136] (see Fig. 17).

Note that many human competitive results were presented at the Humies 2004–2007 competitions (for instance, 11 of the 2004 entries were judged to be human competitive). However, only the very best were awarded medals. So, at the time of writing we estimate that there are at least something of the order of 60 human competitive results obtained by GP. This shows GP's potential as a powerful invention machine.

7.3 Image and Signal Processing

Hampo was one of the first people from industry to consider using GP for signal processing. He evolved algorithms for preprocessing electronic motor vehicle signals for possible use in engine monitoring and control [127]. Several applications of GP for image processing have been for military uses – for example, Tackett evolved algorithms to find tanks in infrared images [370]. Howard evolved program to pick out ships from SAR radar mounted on satellites in space [149] and to locate ground vehicles from airborne photo reconnaissance [150]. He also used GP to process surveillance data for civilian purposes, such as predicting motorway traffic jams from subsurface traffic speed measurements [148]. Using satellite SAR radar, [67] evolved algorithms to find features in polar sea ice. Optical satellite images can also be used for environmental studies [47], and for prospecting for valuable minerals [332].

Alcazar used GP to find recurrent filters (including artificial neural networks – ANN [92]) for one dimensional electronic signals [347]. Local search (simulated annealing or gradient descent) can be used to adjust or fine-tune 'constant' values within the structure created by genetic search [354]. [411]

have used GP to preprocess images, particularly of human faces, to find regions of interest, for subsequent analysis. (See also [382].) A particular strength of GP is its ability to take data from disparate sources [43, 369].

Zhang has been particularly active at evolving programs with GP to visually classify objects (typically coins) [419]. He has also applied GP to human speech [409]. 'Parisian GP' is a system in which the image processing task is split across a swarm of evolving agents ('flies'). In [235, 236] the flies reconstruct three-dimensions from pairs of stereo images. In [235] as the flies buzz around in three-dimensions their position is projected onto the left and right of a pair of stereo images. The fitness function tries to minimize the discrepancy between the two images, thus encouraging the flies to settle on visible surfaces in the 3-D space. So, the true 3–D space is inferred from pairs of 2-D image taken from slightly different positions.

While the likes of `Google` have effectively indexed the written word. For speech and in particular pictures, it has been much less effective. One area where GP might be applied is in automatically indexing images. Some initial steps in this direction are given in [378].

To some extent extracting text from images (OCR) is almost a solved problem. With well formed letters and digits this is now done with near 100% accuracy as a matter of routine. However, many interesting cases remain [58] such as Arabic [182] and oriental languages, handwriting [72, 107, 200, 377] (such as the MNIST examples), other texts [327], and musical scores [317].

The scope for applications of GP to image and signal processing is almost unbounded. A promising area is medical imaging [291]. GP image techniques can also be used with sonar signals [245]. Off-line work on images, includes security and verification. For example, [386] have used GP to detect image watermarks which have been tampered with. Whilst recent work by Zhang has incorporated multi-objective fitness into GP image processing [420].

In 1999 Poli, Cagnoni and others founded the annual *European Workshop on Evolutionary Computation in Image Analysis and Signal Processing* (EvoIASP). *EvoIASP* is held every year along with the *EuroGP*. Whilst not solely dedicated to GP, many GP applications have been presented at *EvoIASP*.

7.4 Financial Trading, Time Series Prediction and Economic Modeling

GP is very widely used in these areas and it is impossible to describe all its applications. It this Section we will hint at just a few areas.

Chen has written more than 60 papers on using GP in finance and economics. Recent papers include modeling of agents in stock markets [52], game

theory [54], evolving trading rules for the S&P 500 [414] and forecasting the Heng-Sheng index [53] (see Chapter 13 of this Compendium).

The *efficient markets hypothesis* is a tenet of economics. It is founded on the idea that everyone in a market has 'perfect information' and acts 'rationally'. If the efficient markets hypothesis held, then everyone would see the same value for items in the market and so agree the same price. Without price differentials, there would be no money to be made from the market itself. Whether it is trading potatoes in northern France or dollars for yen it is clear that traders are not all equal and considerable doubt has been cast on the efficient markets hypothesis. So, people continue to play the stock market. Game theory has been a standard tool used by economists to try to understand markets but is increasingly supplemented by simulations with both human and computerized agents. GP in increasingly being used as part of these simulations of social systems.

Neely and Weller of the US Federal Reserve Bank used GP to study intra-day technical trading of foreign exchange to suggest the market is 'efficient' and found no evidence of excess returns [263, 264, 266, 267]. This negative result was criticized by [244]. Later work by [268] suggested that data after 1995 are consistent with Lo's *adaptive markets hypothesis* rather than the *efficient markets hypothesis*. Note that here GP and computer tools are being used in a novel data-driven approach to try and resolve issues which were previously a matter of dogma.

From a more pragmatic viewpoint, Kaboudan shows GP can forecast international currency exchange rates [164], stocks [165] and stock returns [163]. [383] continue to apply GP to a variety of financial arenas, including: betting, forecasting stock prices [109], studying markets [158] and arbitrage [243]. [15,74,75] and HSBC also use GP in foreign exchange trading. Pillay has used GP in social studies and teaching aids in education, for instance, [289]. As well as trees [187], other types of GP have been used in finance, for example [270].

The *Intl. Conf. on Computing in Economics and Finance* (CEF) has been held every year since 1995. It regularly attracts GP papers, many of which are on-line. In 2007 Brabazon and O'Neill established the *European Workshop on Evolutionary Computation in Finance and Economics* (EvoFIN); *EvoFIN* is held with *EuroGP*.

7.5 Industrial Process Control

Of course most industrialists have little time to spend on academic reporting. A notable exception is Dow Chemical, where Kordon's group has been very active [46,185,254]. [184] describes where industrial GP stands now and how it will progress. Another active collaboration is that between Kovacic and Balic, who have used GP in the computer numerical control of industrial milling and cutting machinery [186]. The partnership of Deschaine and Francone [100] is

most famous for their use of `Discipulus` [99] for detecting bomb fragments and unexploded ordnance UXO [76]. `Discipulus` has been used as an aid in the development of control systems for rubbish incinerators [77].

One of the earliest users of GP in control was Willis' Chemical Engineering group at Newcastle, which used GP to model flow in a plasticating extruder [399]. They also modelled extruding food [251] and control of chemical reactions in continuous stirred tank reactors [342]. Marenbach investigated GP in the control of biotech reactors [242]. [398] surveyed GP applications, including to control. Other GP applications to plastic include [38]. Lewin has applied GP to the control of an integrated circuit fabrication plant [68, 228]. Domingos worked on simulations of nuclear reactors (PWRs to be exact) to devise better ways of preventing xenon oscillations [83]. GP has also been used to identify which state a plant to be controlled is in (in order to decide which of various alternative control laws to apply). For example, Fleming's group in Sheffield used multi-objective GP [141, 329] to reduce the cost of running aircraft jet engines [14, 93]. [4] surveys GP and other AI techniques applied in the electrical power industry.

7.6 Medicine, Biology and Bioinformatics

GP has long been applied to medicine, biology and bioinformatics. Early work by Handley [128] and Koza [191] used GP to make predictions about the behavior and properties of biological systems, principally proteins. Oakley, a practising medical doctor, used GP to model blood flow in toes [276] as part of his long term interests in frostbite.

In 2002 Banzhaf and Foster organized *BioGEC*, the first *GECCO Workshop on Biological Applications of Genetic and Evolutionary Computation. BioGEC* has become a bi-annual feature of the annual *GECCO* conference. Half a year later Marchiori and Corne organized *EvoBio*, the *European Conf. on Evolutionary Computation, Machine Learning and Data Mining in Bioinformatics. EvoBio* is held every year alongside *EuroGP*; GP figures heavily in both *BioGEC* and *EvoBIO*.

GP is often used in data mining. Of particular medical interest are very wide data sets, with many inputs per sample. Examples include infrared spectra [89, 90, 117, 119, 131, 159, 250, 373, 387], single nuclear polymorphisms [26, 320, 345] and Affymetrix GeneChip microarray data [71, 91, 139, 142, 147, 217, 229, 230, 412].

Kell and his colleagues in Aberystwyth have had great success in applying GP widely in bioinformatics (see infrared spectra above and [2, 70, 113, 118, 160, 168–171, 349, 407]). Another very active group is that of Moore and his colleagues at Vanderbilt [261, 262, 325, 326].

Computational chemistry is widely used in the drug industry. The properties of simple molecules can be calculated. However, the interactions between

chemicals which might be used as drugs and medicinal targets within the body are beyond exact calculation. Therefore, there is great interest in the pharmaceutical industry in approximate *in silico* models which attempt to predict either favourable or adverse interactions between proto-drugs and biochemical molecules. Since these are computational models, they can be applied very cheaply in advance of manufacture of chemicals, to decide which of the myriad of chemicals might be worth further study. Potentially such models can make a huge impact both in terms of money and time without being anywhere near 100% correct. Machine learning and GP have both been tried. GP approaches include [21, 27, 43, 95, 114, 120, 131, 134, 199, 351, 388, 393].

7.7 Mixing GP with Other Techniques

GP can be hybridised with other techniques. Iba [152], Nikolaev [269], and Zhang [417] have incorporated information theoretic and minimum description length ideas into GP fitness functions to provide a degree of regularization and so avoid over-fitting (and bloat, see Sect. 9.3). As mentioned in Sect. 5.4 computer language grammars can be incorporated into GP. Indeed Wong [401–403, 405] has had success integrating these with GP. The use of simulated annealing and hill climbing to locally fine tune parts of solutions found by GP was described in Sect. 2.

7.8 GP to Create Searchers and Solvers – Hyper-Heuristics

Hyper-heuristics could simply be defined as 'heuristics to choose other heuristics' [40]. A heuristic is considered as a rule-of-thumb or 'educated guess' that reduces the search required to find a solution. The difference between meta-heuristics and hyper-heuristics is that the former operate directly on the problem search space with the goal of finding optimal or near-optimal solutions. The latter, instead, operate on the heuristics search space (which consists of the heuristics used to solve the target problem). The goal then is finding or generating high-quality heuristics for a problem, for a certain class of instances of a problem, or even for a particular instance.

GP has been very successfully used as a hyper-heuristic. For example, GP has evolved competitive SAT solvers [19, 20, 103, 177], state-of-the-art or better than state-of-the-art bin packing algorithms [41, 42, 313], particle swarm optimizers [310, 311], evolutionary algorithms [277], and travelling salesman problem solvers [172–174, 278].

7.9 Artistic

Computers have long been used to create purely aesthetic artifacts. Much of today's computer art tends to ape traditional drawing and painting, producing static pictures on a computer monitor. However, the immediate advantage

of the computer screen – movement – can also exploited. In both cases EC can and has been exploited. Indeed with evolution's capacity for unlimited variation, EC offers the artist the scope to produce ever changing works. Some artists have also worked with sound.

The use of GP in computer art can be traced back at least to the work of Sims [353] and Latham. Jacob provides many examples [155,156]. Since 2003, *EvoMUSART* has been held every year with *EuroGP*. McCormack considers the recent state of play in evolutionary art and music [249]. Many recent techniques are described in [241].

Evolutionary music [379] has been dominated by Jazz [359], which is not to everyone's taste; an exception is Bach [94]. Most approaches to evolving music have made at least some use of interactive evolution [371] in which the fitness of programs is provided by users, often via the Internet [5,49]. The limitation is almost always finding enough people willing to participate [211]. Funes reports experiments which attracted thousands of people via the Internet who were entertained by evolved `Tron` players [105]. Costelloe tried to reduce the human burden in [62]; algorithmic approaches are also possible [59,153].

One of the sorrows of AI is that as soon as it works it stops being AI (and celebrated as such) and becomes computer engineering. For example, the use of computer generated images has recently become cost effective and is widely used in Hollywood. One of the standard state-of-the-art techniques is the use of Reynold's swarming 'boids' [321] to create animations of large numbers of rapidly moving animals. This was first used in *Cliffhanger* (1993) to animate a cloud of bats. Its use is now common place (herds of wildebeest, schooling fish, and the like); in 1997 Craig was awarded an Oscar.

7.10 Entertainment and Computer Games

Today the major usage of computers is interactive games [315]. There has been a little work on incorporating artificial intelligence into mainstream commercial games. The software owners are not keen on explaining exactly how much AI they use or giving away sensitive information on how they use AI. Work on GP and games includes [16,389]. Since 2004 the annual *CEC Conference* has included sessions on EC in games. After chairing the *IEEE Symposium on Computational Intelligence and Games 2005* at Essex University, Lucas founded the IEEE Computational Intelligence Society's Technical Committee on Games. GP features heavily in the Games TC's activities, for example Othello, Poker, Backgammon, Draughts, Chess, Ms Pac-Man, robotic football and radio controlled model car racing.

7.11 Where can we Expect GP to Do Well?

GP and other EC 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 to the problem is a major part of the problem.
- Large amounts of primary data requiring examination, classification, and integration is 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.

The best predictor of future performance is the past. So, we should expect GP to continue to be successful in application domains with these features.

8 Tricks of the Trade

8.1 Getting Started

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

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

Let us start from question 1. A variety of sources of information about GP are available (many of which are listed in the Resources Appendix). Consulting information available on the Web is certainly a good way to get quick answers for a 'newbie' who wants to know what GP is. These answers, however, will often be too shallow for someone who really wants to then apply GP to solve practical problems. People in this position should probably invest some time going through more detailed accounts, such [25, 188, 222] or some of the other books in the Resources Appendix. Technical papers may be the next stage. The literature on GP is now quite extensive. So, although this is easily accessible thanks to the complete online bibliography (http://www.cs.bham.ac.uk/~wbl/biblio/), newcomers will often need to be selective in what they read. The objective here may be different for different types of readers. Practitioners should probably identify and read only papers which deal with the same problem they are interested in. Researchers and PhD students interested in developing a deeper understanding of GP should also make sure they identify and read as many seminal papers as possible,

including papers or books on empirical and theoretical studies on the inner mechanisms and behavior of GP. These are frequently cited in other papers and so can easily be identified.

The answer to question 2 depends on the particular experience and background of the questioner. Implementing a simple GP system from scratch is certainly an excellent way to make sure one really understands the mechanics of GP. In addition to being an exceptionally useful exercise, this will always result in programmers knowing their systems so well that they will have no problems customizing them for specific purposes (for example, adding new, application specific genetic operators or implementing unusual, knowledge-based initialization strategies). All of this, however, requires reasonable programming skills and the will to thoroughly test the resulting system until it fully behaves as expected. If the skills or the time are not available, then the best way to get a working GP application is to retrieve one of the many public-domain GP implementations and adapt this for the user's purposes. This process is faster, and good implementations are often quite robust, efficient, well-documented and comprehensive. The small price to pay is the need to study the available documentation and examples. These often explain also how to modify the 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 of the system and a substantial amount of trial and error. Good, publicly-available GP implementations include: `Lil-GP`, `ECJ`, and `DGPC`.

While perhaps to some not as exciting as coding or running GP, a through search of the literature can avoid 're-inventing the wheel'.

8.2 Presenting Results

It is so obvious that it is easy to forget one major advantage of GP: we create *visible* programs. That is, the way they work is accessible. This need not be the case with other approaches. So, when presenting GP results, as a matter of routine one should perhaps always make a comprehensible slide or figure which contains the whole evolved program,[2] trimming unneeded details (such as removing excess significant digits) and combining constant terms. Naturally, after cleaning up the answer, one should make sure the program still works.

If one's goal is to find a comprehensible model, in practice it must be small. A large model will not only be difficult to understand but also may over-fit the training data [112]. For this reason (and possibly others), one should use one of the anti-bloat mechanisms described in Sect. 9.3.

There are methods to automatically simplify expressions (for example, in `Mathematica` and `Emacs`). However, since in general there is an exponentially

[2] The program `Lisp2dot` can be of help in this.

large number of equivalent expressions, automatic simplification is hard. Another way is to use GP. After GP has found a suitable but large model, one can continue evolution changing the fitness function to include a second objective: that the model be as small as possible [203]. GP can then trim the trees but ensure the evolved program still fits the training data.

It is important to use the language that one's customers, audience or readers use. For example, if the fact that GP discovers a particular chemical is important, one should make this fact standout, say by using colours. Also, GP's answer may have evolved as a tree but, if the customers use Microsoft `Excel`, it may be worthwhile translating the tree into a spreadsheet formula.

Also, one should try to discover how the customers intend to validate GP's answer. Do not let them invent some totally new data which has nothing to do with the data they supplied for training ('just to see how well it does...'). Avoid customers with contrived data. GP is not god, it knows nothing about things it has not seen. At the same time users should be scrupulous about their own use of holdout data. GP is a very powerful machine learning technique. With this comes the ever present danger of over-fitting. One should never allow performance on data reserved for validation to be be used to choose which answer to present to the customer.

8.3 Reducing Fitness Evaluations/Increasing their Effectiveness

While admirers of linear GP will suggest that machine code GP is the ultimate in speed, tree GP can be made faster in a number of ways. The first is to reduce the number of times a tree is evaluated. Many applications find the fitness of trees by running them on multiple training examples. However, ultimately the point of fitness evaluation is to make a binary decision: does this individual get a child or not. Indeed usually a noisy selection technique is used such as roulette wheel, SUS [22], or tournament selection. Stochastic selection is an essential part of genetic search but it necessarily injects noise into the vital decision of which points in the search to proceed from and which to abandon. The overwhelming proportion of GP effort (or indeed any EC technique) goes into adjusting the probability of the binary decision as to whether each individual in the population should be allowed to reproduce or not. If a program has already demonstrated it works very badly compared to the rest of the population on a fraction of the available training data, it is likely not to have children. Conversely, if it has already exceeded many programs in the population after being tested on only a fraction of the training set, it is likely to have a child [203]. In either case, it is apparent that we do not need to run it on the remaining training examples. Teller and Andre developed this idea into an effective algorithm [376].

As well as the computational cost, there are other aspects of using all the training data all the time. It gives rise to a static fitness function. Arguably

this tends to evolve the population into a cul-de-sac where the population is dominated by offspring of a single initial program which did well of some fraction of the training data but was unable to fit others. A static fitness function can easily have the effect that the other good programs which perhaps did well on other parts of the training data get lower fitness scores and fewer children.

With high selection pressure, it takes surprisingly little time for the best individual to dominate the whole population. Goldberg calls this the 'take over time' [115]. This can be made quite formal [31, 85]. However, for tournament selection, a simple rule of thumb is often sufficient. If T is the tournament size, about $\log_T$(Pop size) generations are needed for the whole population to become descendants of a single individual. For example, if we use binary tournaments ($T = 2$), then 'take over' will require about ten generations for a population of 1,024. Alternatively, if we have a population of one million (10^6) and use ten individuals in each tournament ($T = 10$), then after about six generations more or less everyone will have the same great_6 great_5 great_4 great_3 grand_2 mother_1.

Gathercole investigated a number of ways of changing which training examples to use as the GP progressed [110,111]. (Siegel proposed a rather different implementation in [352].) This juggles a number of interacting effects. Firstly, by using only a subset of the available data, the GP fitness evaluation takes less time. Secondly, by changing which examples are being used, the evolving population sees more of the training data and, so, is less liable to over fit a fraction of it. Thirdly, by randomly changing the fitness function, it becomes more difficult for evolution to produce an over specialized individual which takes over the population at the expense of solutions which are viable on other parts of the training data. Dynamic Subset Selection (DSS) appears to have been the most successful of Gathercole's suggested algorithms. It has been incorporated into `Discipulus`. Indeed a huge data mining application [66] recently used DSS.

Where each fitness evaluation may take a long time, it may be attractive to interrupt a long running program in order to let others run. In GP systems which allow recursion or contain iterative elements [36, 203, 400, 404] it is common to enforce a time limit, a limit on the number of instructions executed, or a bound on the number of times a loop is executed. Maxwell proposed [248] a solution to the question of what fitness to we give to a program we have interrupted. He allowed each program in the population a quantum of CPU time. When the program uses up its quantum it is check-pointed. When the program is check-pointed, sufficient information (principally the program counter and stack) is saved so that it can be restarted from where it got to later. (Many multi-tasking operating systems do something similar.) In Maxwell's system, he assumed the program gained fitness as it ran. For example, each time is correctly processes a fitness case, its fitness is incremented. So the fitness of

a program is defined while it is running. Tournament selection is then performed. If all members of the tournament have used the same number of CPU quanta, then the program which is fitter is the winner. However, if a program has used less CPU than the others (and has a lower fitness) then it is restarted from where it was and is run until it has used as much CPU as the others. Then fitnesses are compared in the normal way.

Teller had a similar but slightly simpler approach: everyone in the population was run for the same amount of time. When the allotted time elapses the program is aborted and an answer extracted from it, regardless of whether it was ready or not; he called this an 'any time' approach [374]. This suits graph or linear GP, where it is easy to designate a register as the output register. The answer can be extracted from this register or from an indexed memory cell at any point (including whilst the programming is running). Other any time approaches include [220, 360].

A simple technique to speed up the evaluation of complex fitness functions is to organize the fitness function into stages of progressively increasing computational cost. Individuals are evaluated stage by stage. Each stage contributes to the overall fitness of a program. However, individuals need to reach a minimum fitness value in each stage in order for them to be allowed to progress to the next stage and acquire further fitness. Often different stages represent different requirements and constraints imposed on solution.

Recently, a sophisticated technique, called *backward chaining GP*, has been proposed [297, 301–303] that can radically reduce the number of fitness evaluations in runs of GP (and other EAs) using tournament selection with small tournament sizes. Tournament selection randomly draws programs from the population to construct tournaments, the winners of which are then selected. Although this process is repeated many times in each generation, when the tournaments are small there is a significant probability that an individual in the current generation is never chosen to become a member of any tournament. By reordering the way operations are performed in GP, backward chaining GP exploits this not only to avoid the calculation of individuals that are never sampled, but also to achieve higher fitness sooner.

8.4 Co-Evolution

One way of viewing DSS is as automated co-evolution. In co-evolution there are multiple evolving species (typically two) whose fitness depends upon the other species. (Of course, like DSS, co-evolution can be applied to linear and other types of GP as well as tree GP.) One attraction of co-evolution is that it effectively produces the fitness function for us. There have been many successful applications of co-evolution [16, 35, 39, 48, 82, 108, 140, 338, 346, 400], however it complicates the already complex phenomena taking place in the presence of dynamic fitness functions still further. Therefore, somewhat reluctantly, at present it appears to be beneficial to use co-evolution only if an

application really requires it. Co-evolution may suffer from unstable populations. This can occur in nature, oscillations in Canadian Lynx and Snowshoe Hare populations being a famous example. There are various 'hall of fame' techniques [106], which try to damp down oscillations and prevent evolution driving competing species in circles.

8.5 Reducing Cost of Fitness with Caches

In computer hardware it is common to use data caches which automatically hold copies of data locally in order to avoid the delays associated with fetching it from disk or over a network every time it is needed. This can work well where a small amount of data is needed many times over a short interval. Caches can also be used to store results of calculations, thereby avoiding the re-calculation of data [129]. GP populations have enormous amounts of common code [203, 215, 220]. This is after all how genetic search works: it promotes the genetic material of fit individuals. So, typically in each generation we see many copies of successful code. In a typical GP system, but by no means all GP systems, each subtree has no side-effects. This means its results pass through its root node in a well organized and easy to understand fashion. Thus, if we remember a subtree's inputs and output when it was run before, we can avoid re-executing code whenever we are required to run the subtree again. Note that this is true irrespective of whether we need to run the same subtree inside a different individual or at a different time (namely, a later generation). Thus, if we stored the output with the root node, we need only run the subtree once, for a given set of inputs. Whenever the interpreter comes to evaluate the subtree, it needs only to check if the root contains a cache of the values the interpreter calculated last time, thus saving considerable computation time. However, there is a problem: not only must the answer be stored, but the interpreter needs to know that the subtree's inputs are the same too.

The common practices of GP come to our aid here. Usually every tree in the population is run on exactly the same inputs for each of the fitness cases. Thus, for a cache to work, the interpreter does not need to know in detail which inputs the subtree has or their exact values corresponding to every value calculated by the subtree. It need only know which of the fixed set of test cases was used.

A simple cache implementation is to store a vector of values returned by each subtree. The vector is as long as the number of test cases. Whenever a subtree is created (namely, in the initial generation, by crossover or by mutations) the interpreter is run and the cache of values for its root node is set. Note this is recursive, so caches can also be calculated for subtrees within it at the same time. Now when the interpreter is run and comes to a subtree's root node, it will know which test case it is running and instead of interpreting the subtree it simply retrieves the value it calculated using the

test case's number as an index into the cache vector. This could be many generations after the subtree was originally created.

If a subtree is created by mutation, then its cache of values will be initially empty and will have to be calculated. However, this costs no more than without caches.

When subtrees are crossed over the subtree's cache remains valid and so cache values can be crossed over like the code.

When code is inserted into an existing tree, be it by mutation or crossover, the chance that the new code behaves identically to the old code is normally very small. This means the caches of every node between the new code and the root node may be invalid. The simplest thing is to re-evaluate them all. This sounds expensive, but remember the caches in all the other parts of the individual remain valid and so can be used when the cache above them is re-evaluated. Thus, in effect, if the crossed over code is inserted at level-d, only d nodes need to be evaluated. Recent analysis [57,81,222,312] has shown that GP trees tend not to have symmetric shapes, and many leafs are very close to the root. Thus in theory (and in practice) considerable computational saving can be made by using fitness caches. Sutherland is perhaps the best known GP system which has implemented fitness caches [253]. As well as the original DAG implementation [129]; other work includes [57, 166, 410].

In [203] we used fitness caches in evolved trees with side effects by exploiting syntax rules about where in the code the side-effects could lie. The whole question of monitoring how effective individual caches are, what their hit-rates are, and so on, has been little explored. In practice, in many common GP systems, impressive savings have been made by simple implementations, with little monitoring and rudimentary garbage collection. While it is possible to use hashing schemes to efficiently find common code, in practice assuming that common code only arises because it was inherited from the same location (for instance, by crossing over) is sufficient.

8.6 GP Running in Parallel

In contrast to much of computer science, EC can be readily run on parallel computer hardware; indeed it is 'embarrassingly parallel' [7]. For example, when Turton ran GP on a Cray supercomputer he obtained about 30% of its theoretical peak performance, embarrassing his 'supercomputer savvy' colleagues who rarely got better than a few percent out of it [280].

There are two important aspects of parallel evolutionary algorithms. These are equally important but often confused. The first is the traditional aspect of parallel computing. We port an existing algorithm onto a supercomputer so that it runs faster. The second aspect comes from the biological inspiration for EC.

In Nature everything happens in parallel. Individuals succeed or not in producing and raising children at the same time as other members of their species. The individuals are spread across oceans, lakes, rivers, plains, forests, mountain chains, and the like. It was this geographic spread that led Wright to propose that geography and changes to it are of great importance to the formation of new species and so to natural evolution as a whole [408].

While in Nature geographically distributed populations are a necessity, in EC we have a choice. We can run GP on parallel hardware so as to speed up runs, or we can distribute GP populations over geographies so as obtain some of the benefits it brings to natural evolution. In the following we will discuss both ideas. It is important to note, however, that one does not need to use parallel hardware to use geographically distributed GP populations. Although parallel hardware naturally lends itself to realize *physically-distributed* populations, one can obtain similar benefits by using *logically-distributed* populations in a single machine.

Master-Slave GP

If the objective is purely to speed up runs, we may want our GP to work exactly the same as it did on a single computer. This is possible, but to achieve it we have to be very careful to ensure that even if some parts of the population are evaluated quicker, that parallelization does not change how we do selection and which GP individual crosses over with the other. Probably the easiest way to implement this is the master-slave model.

In the master-slave model [285], breeding, selection, crossover, mutation and so on are exactly as on a single computer, and only fitness evaluation is spread across a network of computers. Each GP individual and its fitness cases are sent across the network to a compute node. The central node waits for it to return the individual's fitness. Since individuals and fitness values are small, this can be quite efficient. The central node is an obvious bottleneck. Also, a slow compute node or a lengthy fitness case will slow down the whole GP population, since eventually its result will be needed before moving onto the next generation.

Geographically Distributed GP

As we have seen, unless some type of synchronization or check pointing is imposed, say at the end of each generation, the parallel GP will not be running the same algorithm as the single node version, and, so, it will almost certainly produce different answers. If the population is divided up into sub-populations (known as *demes* [60, 80, 203]) and the exchange of individuals among populations is limited (both in terms of how many individuals are allowed to migrate per generation and a geography that constraints which populations can communicate with which), then parallelization can bring benefits similar

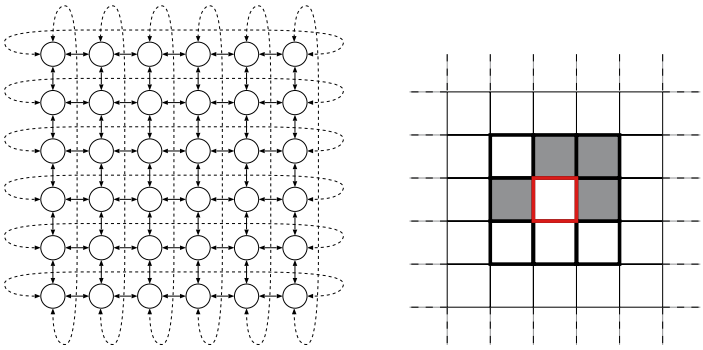

Fig. 18. Spatially structured GP populations. (**a**) Toroidal grid of demes, where each deme (node) contains a sub-population, and demes periodically exchange a small group of high-fitness individuals using a grid of communication channels. (**b**) Fine-grained distributed GP, where each grid cell contains one individual and where the selection of a mating partner for the individual in the centre cell is performed by executing a tournament among randomly selected individuals (for instance, the individuals shaded) in its 3×3 neighbourhood

to those found in Nature by [408]. For example, it may be that with limited migration between compute nodes, the evolved populations on adjacent nodes will diverge and that this increased diversity may lead to better solutions.

When Koza first started using GP on a network of Transputers [6], Andre experimentally determined the best immigration rate for their problem. He suggested Transputers arranged in an asynchronous 2-D toroidal square grid (such as the one in Fig. 18a) should exchange 2% of their population with their four neighbours.

Densely connected grids have been widely adopted in parallel GP. Usually they allow innovative partial solutions to quickly spread. However, the GA community reported better results from less connected topologies, such as arranging the compute node's populations in a ring, so that they could transport genes only between between themselves and their two neighbours [365]. Potter argues in favour of spatial separation in populations (see Fig. 18b) [314]. Goldberg also suggests low migration rates [116]. In [396], Whitley includes some guidance on parallel GAs.

While many have glanced enviously at Koza's 1000 node Beowulf [368], a supercomputer [29, 162] is often not necessary. Many businesses and research centres leave computers permanently switched on. During the night their computational resources tend to be wasted. This computing power can easily and efficiently be used to execute distributed GP runs overnight. Typically GP does not demand a high performance bus to interconnect the compute nodes, and, so, existing office Ethernet LANs are often sufficient. Whilst parallel

Fig. 19. A global population [213].; the straight lines show connections between major sites in a continuously evolving L-System

GP systems can be implemented using `MPI` [391] or `PVM` [96], the use of such tools is not necessary: simple Unix commands and port-to-port HTTP is sufficient [307]. The population can be split and stored on modest computers. With only infrequent interchange of parts of the population or fitness values little bandwidth is needed. Indeed a global population spread via the Internet [213], *á la* `seti@home`, is perfectly feasible [56]. (See Fig. 19). Other parallel GPs include [6, 44, 50, 63, 97, 98, 125, 183, 232, 246, 334, 335, 372].

GP Running on GPUs

Modern PC graphics cards contain powerful Graphics Processing Units (GPUs) including a large number of computing components. For example, it is not atypical to have 128 streaming processors on a single PCI graphics card. In the last few years there has been an explosion of interest in porting scientific or general purpose computation to mass market graphics cards [286].

Indeed, the principal manufactures (`nVidia` and `ATI`) claim faster than Moore's Law increase in performance, suggesting that GPU floating point performance will continue to double every twelve months, rather than the 18–24 months observed [260] for electronic circuits in general and personal computer CPUs in particular. In fact, the apparent failure of PC CPUs to keep up with Moore's law in the last few years makes GPU computing even more attractive. Even today's bottom of the range GPUs greatly exceed the floating point performance of their hosts' CPU. However, this speed comes at a price, since GPUs provide a restricted type of parallel processing, often referred to a single instruction multiple data (SIMD) or single program multiple data

(SPMD). Each of the many processors simultaneously runs the same program on different data items.

There have been a few GP experiments with GPUs [55, 86, 130, 216, 218, 237, 255, 319]. So far, in GP, GPUs have just been used for fitness evaluation. Harding used the Microsoft research GPU development `Direct X` tools to allow him to compile a whole population of Cartesian GP network programs into a GPU program [132] which was loaded onto his Laptop's GPU in order to run fitness cases. We used [216, 218] a SIMD interpreter [162] written in C++ using RapidMind's `GCC OpenGL` framework to simultaneously run up to a quarter of a million GP trees on an `nVidia` GPU. A conventional tree GP S-expression can be linearized. We used used reverse polish notation (RPN) – that is, post fix notation – rather than pre-fix notation. RPN avoids recursive calls in the interpreter [216]. Only small modifications are needed to do crossover and mutation so that they act directly on the RPN expressions. This means the same representation is used on both the host and the GPU. In both Cartesian and tree GP the genetic operations are done by the host CPU. Wong showed, for a genetic algorithm, these too can be done by the GPU [406].

Although each of the GPU's processors may be individually quite fast and the manufacturers claim huge aggregate FLOP ratings, the GPUs are optimized for graphics work. In practice it is hard to keep all the processors fully loaded. Nevertheless 30 GFLOPS has been achieved [218]. Given the differences in CPU and GPU architectures and clock speeds, often the speedup from using a GPU rather than the host CPU is the most useful statistic. This is obviously determined by many factors, including the relative importance of amount of computation and size of data. The measured RPN tree speedups were 7.6 [218] and 12.6 [216].

8.7 GP Trouble-Shooting

A number of practical recommendations for GP work can be made. To a large extent the advice in [181] and [188] remains sound. However, we also suggest:

- GP populations should be closely studied as they evolve. There are several properties that can be easily measured which give indications of problems:
  - *Frequency of primitives.* Recognizing when a primitive has been completely lost from the population (or its frequency has fallen to a low level, consistent with the mutation rate) may help to diagnose problems.
  - *Population variety.* If the variety – the number of distinct individuals in the population – falls below 90% of the population size, this indicates there may be a problem. However, a high variety does not mean the reverse. GP populations often contain introns, and so programs which are not identical may behave identically. Being different, these

individuals contribute to a high variety, that is a high variety need not indicate all is well. Measuring phenotypic variation (that is, diversity of behavior) may also be useful.

- Measures should be taken to encourage population diversity. Panmictic steady-state populations with tournament selection, reproduction and crossover may converge too readily.[3] The above-mentioned metrics may indicate if this is happening in a particular case. Possible solutions include:
  - Not using the reproduction operator.
  - Addition of one or more mutation operators.
  - Smaller tournament sizes and/or using uniform random selection (instead of the standard negative tournaments) to decide which individuals to remove from the population. Naturally, the latter means the selection scheme is no longer elitist; it may be worthwhile forcing it to be elitist.
  - Splitting large populations into semi-isolated demes.[4]
  - Using fitness sharing to encourage the formation of many fitness niches.
- Use of *fitness caches* (either when executing an individual or between ancestors and children) can reduce run time and may repay the additional work involved with using them.
- Where GP run time is long, it is important to periodically save the current state of the run. Should the system crash, the run can be restarted from part way through rather than at the start. Care should be taken to save the entire state, so restarting a run does not introduce any unknown variation. The bulk of the state to be saved is the current population. This can be compressed, for example by using `gzip`. While compression can add a few percent to run time, reductions in disk space to less than one bit per primitive in the population have been achieved.

9 Genetic Programming Theory

Most of this Chapter is about the mechanics of GP and its practical use for solving problems. We have looked at GP from a problem-solving and engineering point of view. However, GP is a non-deterministic searcher and, so, its behavior varies from run to run. It is also a complex adaptive system which sometimes shows complex and unexpected behaviors (such as bloat). So, it

[3] In a panmictic population no mating restrictions are imposed as to which individual mates with which.

[4] What is meant by a 'large population' has changed over time. In the early days of GP populations of 1,000 or more could be considered large. However, CPU speeds and computer memory have increased exponentially over time. So, at the time of writing it is not unusual to see populations of hundred of thousands or millions of individuals being used in the solution of hard problems. Research indicates that there are benefits in splitting populations into demes even for much smaller populations.

is only natural to be interested in GP also from the scientific point of view. That is, we want to understand why can GP solve problems, how it does it, what goes wrong when it cannot, what are the reasons for certain undesirable behaviors, what can we do to get rid of them without introducing new (and perhaps even less desirable) problems, and so on.

GP is a search technique that explores the space of computer programs. The search for solutions to a problem starts from a group of points (random programs) in this search space. Those points that are 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 stopping criterion is satisfied. If we could *visualize* this search, we would often find that initially the population looks like a cloud of randomly scattered points, but that, generation after generation, this cloud changes shape and moves in the search space. Because GP is a stochastic search technique, in different runs we would observe different trajectories. These, however, would show clear regularities which would provide us with a deep understanding of how the algorithm is searching the program space for the solutions. We would probably readily see, for example, why GP is successful in finding solutions in certain runs, and unsuccessful in 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 GP.

9.1 Mathematical Models

In this situation, in order to gain an understanding of the behavior of a GP system one can perform many real runs and record the variations of certain numerical descriptors (like the average fitness or the average size of the programs in the population at each generation, the average number of inactive nodes, the average difference between parent and offspring fitness, and so on). Then, one can try to suggest explanations about the behavior of the system which are compatible with (and could explain) the empirical observations. This exercise is very error prone, though, because a genetic programming system is a complex adaptive system with 'zillions' of degrees-of-freedom. So, any small number of statistical descriptors is likely to be able to capture only a tiny fraction of the complexities of such a system. This is why in order to understand and predict the behavior of GP (and indeed of most other evolutionary algorithms) in precise terms we need to define and then study *mathematical models of evolutionary search*.

Schema theories are among the oldest and the best known models of evolutionary algorithms [145, 397]. 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 GA schema theory [366, 367] 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.[5]

The theory of schemata in GP has had a difficult childhood. Some excellent early efforts led to different worst-case-scenario schema theorems [3, 188, 283, 298, 330, 394]. Only very recently have the first exact schema theories become available [293–295] which give exact formulations (rather than lower bounds) for the expected number of individuals sampling a schema at the next generation. Initially [293, 295], these exact theories were only applicable to GP with one-point crossover (see Sect. 2.4). However, more recently they have been extended to the class of homologous crossovers [309] and to virtually all types of crossovers that swap subtrees [305, 306], including standard GP crossover with and without uniform selection of the crossover points, one-point crossover, context-preserving crossover and size-fair crossover which have been described in Sect. 2.4, as well as more constrained forms of crossover such as strongly-typed GP crossover (see Sect. 5.4), and many others.

9.2 Search Spaces

Exact schema-based models of GP are probabilistic descriptions of the operations of selection, reproduction, crossover, and mutation. They make it explicit how these operations determine the areas of the program space that will be sampled by GP and with which probability. However, these models treat the fitness function as a black box. That is, there is no notion of the fact that in GP, unlike other evolutionary techniques, the fitness function involves the execution of computer programs with different input data. In other words, schema theories do not tell us how fitness is distributed in the search space.

The characterization of the space of computer programs explored by GP has been another main topic of theoretical research [222].[6] 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. For example, in Fig. 20 we plot the proportion of binary program trees composed of NAND gates which implement each of the $2^{2^3} = 256$ Boolean functions of three inputs.

[5] Other models of evolutionary algorithms exist, such those based on Markov chain theory (for example [69, 271]) or on statistical mechanics (for instance, [316]). Only Markov models [258, 308, 309] have been applied to GP, but they are not as developed as schema theory.

[6] Of course results describing the space of all possible programs are widely applicable, not only to GP and other search-based automatic programming techniques, but also to many other areas ranging from software engineering to theoretical computer science.

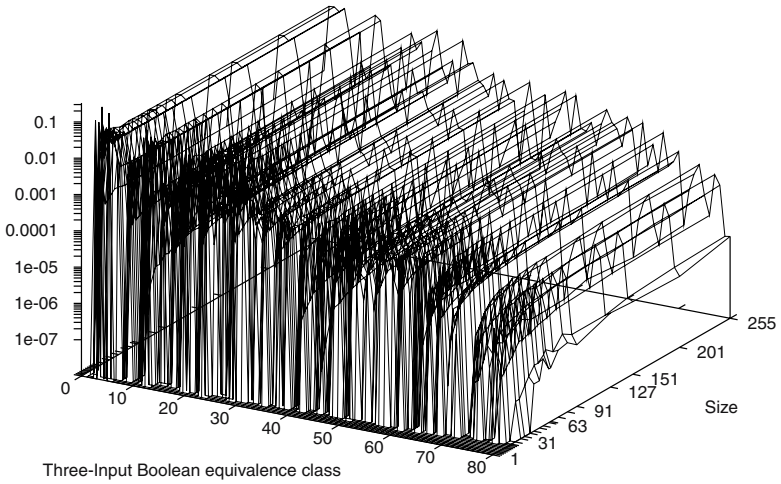

Fig. 20. Proportion of NAND-trees that yield each three-input functions; as circuit size increases the distribution approaches a limit

Notice how, as the length of programs increases, the proportion of programs implementing each function approaches a limit. This does not happen by accident. There is a very substantial body of empirical evidence indicating that this happens in a variety of other systems. In fact, we have also been able to prove mathematically these convergence results for two important forms of programs: Lisp (tree-like) S-expressions (without side effects) and machine code programs without loops [207–210, 212, 222]. Also, similar results were derived for: (a) cyclic (increment, decrement and NOP), (b) bit flip computer, (flip bit and NOP), (c) any non-reversible computer, (d) any reversible computer, (e) CCNOT (Toffoli gate) computer, (f) quantum computers, (g) the 'average' computer and h) AND, NAND, OR, NOR expressions (however, these are not Turing complete).

Recently, we started extending our results to Turing complete machine code programs [304]. We considered a simple but realistic Turing complete machine code language, T7. It includes: directly accessed bit addressable memory, an addition operator, an unconditional jump, a conditional branch and four copy instructions. We performed a mathematical analysis of the halting process based on a Markov chain model of program execution and halting. The model can be used to estimate, for any given program length, important quantities, such as the halting probability and the run time of halting programs. This showed a scaling law indicating that the halting probability for programs of length L is of order $1/\sqrt{L}$, while the expected number of instructions executed by halting programs is of order $\sqrt{L}$. In contrast to many proposed Markov models, this can be done very efficiently, making it possible to compute these quantities for programs of tens of million instructions in a few minutes. Experimental results confirmed the theory.

9.3 Bloat

There are a certain number of limits in GP: bloat, limited modularity of evolved solutions and limited scalability of GP as the problem size increases. We briefly discuss the main one, bloat, below.

Starting in the early 1990s, researchers began to notice that in addition to progressively increasing their mean and best fitness, GP populations also showed certain other dynamics. In particular, it was noted that very often the average size (number of nodes) of the programs in a population, after a certain number of generations in which it was largely static, at some point would start growing at a rapid pace. Typically the increase in program size was not accompanied by any corresponding increase in fitness. The origin of this phenomenon, which is know as *bloat*, has effectively been a mystery for over a decade.

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 (a situation which often arises in continuous symbolic regression problems). So, we should not equate bloat with growth. We should only talk of bloat when there is growth without (significant) return in terms of fitness.

Because of its surprising nature and of its practical effects (large programs are hard to interpret, may have poor generalization and are computationally expensive to evolve and later use), bloat has been a subject of intense study in GP. As a result, many theories have been proposed to explain bloat: replication accuracy theory, removal bias theory, nature of program search spaces theory, and so on. Unfortunately, only recently we have started understanding the deep reasons for bloat. So, there is a great deal of confusion in the field as to the reasons of (and the remedies for) bloat. For many people bloat is still a puzzle.

Let us briefly review these theories:

Replication accuracy theory [252]: This theory states that the success of a GP individual depends on its ability to have offspring that are functionally similar to the parent. So, GP evolves towards (bloated) representations that increase replication accuracy.

Removal bias theory [356]: 'Inactive code' (code that is not executed, or is executed but its output is then discarded) in a GP tree is low in the tree, forming smaller-than-average-size subtrees. Crossover events excising inactive subtrees produce offspring with the same fitness as their parents. On average the inserted subtree is bigger than the excised one, so such offspring are bigger than average.

Nature of program search spaces theory [221,225]: Above a certain size, the distribution of fitnesses does not vary with size. Since there are more long programs, the number of long programs of a given fitness is greater than the number of short programs of the same fitness. Over time GP samples longer and longer programs simply because there are more of them.

Crossover bias theory [81,312]: On average, each application of subtree crossover removes as much genetic material as it inserts. So, crossover in itself does not produce growth or shrinkage. However, while the mean program size is unaffected, other moments of the distribution are. In particular, we know that crossover pushes the population towards a particular distribution of program sizes (a Lagrange distribution of the second kind), where small programs have a much higher frequency than longer ones. For example, crossover generates is a very high proportion of single-node individuals. In virtually all problems of practical interest, very small programs have no chance of solving the problem. As a result, programs of above average length have a selective advantage over programs of below average length. Consequently, the mean program size increases.

Several effective techniques to control bloat have been proposed [225,355]. For example, size fair crossover or size fair mutation [64,206], Tarpeian bloat control [296], parsimony pressure [416–418], or using many runs each lasting only a few generations. Generally the use of multiple genetic operations, each making a small change, seems to help [11,281]. There are also several mutation operators that may help control the average tree size in the population while still introducing new genetic material. [178] proposes a mutation operator which prevents the offspring's depth being more then 15% larger than its parent. [202] proposes two mutation operators in which the new random subtree is on average the same size as the code it replaces. In Hoist mutation [180] the new subtree is selected from the subtree being removed from the parent, guaranteeing that the new program will be smaller than its parent. Shrink mutation [9] is a special case of subtree mutation where the randomly chosen subtree is replaced by a randomly chosen terminal.

10 Conclusions

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

> "There is the genetical or evolutionary search by which a combination of genes is looked for, the criterion being the survival value." [384]

Turing did not specify how to conduct the 'genetical or evolutionary search' for machine intelligence. In particular, he did not mention the idea of a population-based parallel search in conjunction with sexual recombination

(crossover) as described in John Holland's 1975 book [146]. However, in his 1950 paper 'Computing Machinery and Intelligence', he 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:
>
> 'Structure of the child machine' = Hereditary material
> 'Changes of the child machine' = Mutations
> 'Natural selection' = Judgement of the experimenter" [385]

In other words, Turing perceived 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 (that is, 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 Samuel's position statement quoted in the introduction of this chapter. The eight criteria for human competitiveness we discussed in Sect. 7.2 are motivated by the same goals.

At present GP is unable 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 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 Sect. 7.2). These are a small step towards fulfilling Turing and Samuel's dreams, but they are also early signs of things to come. It is, indeed, arguable that in a few years' time GP will be able to *routinely* and *competently* solve important problems for us in a variety of application domains with human-competitive performance. Genetic programming will then become an essential collaborator for many human activities. This, we believe, will be a remarkable step forward towards achieving true, human-competitive machine intelligence.

Acknowledgements

Some of the material in this book chapter has previously appeared in a more extended form in R. Poli, W.B. Langdon, N.F. McPhee, *A Field Guide to Genetic Programming*, lulu.com, 2008. Permission to reproduce it here has been granted by the copyright holders. We would like to thank Rick Riolo, Matthew Walker, Christian Gagne, Bob McKay, Giovanni Pazienza and Lee Spector for their timely assistance.

References

1. Al-Sakran SH, Koza JR, Jones LW (2005) Automated re-invention of a previously patented optical lens system using genetic programming. In: Keijzer M, Tettamanzi A, Collet P, van Hemert JI, Tomassini M (eds) Proceedings of the 8th European Conference on Genetic Programming, Springer, Lausanne, Switzerland, Lecture Notes in Computer Science, vol 3447, pp 25–37, URL http://springerlink.metapress.com/openurl.asp?genre=article&i%ssn=0302-9743&volume=3447&spage=25
2. Allen J, Davey HM, Broadhurst D, Heald JK, Rowland JJ, Oliver SG, Kell DB (2003) High-throughput classification of yeast mutants for functional genomics using metabolic footprinting. Nature Biotechnology 21(6):692–696, DOI doi:10.1038/nbt823, URL http://dbkgroup.org/Papers/NatureBiotechnology21(692-696).pdf
3. Altenberg L (1994) Emergent phenomena in genetic programming. In: Sebald AV, Fogel LJ (eds) Evolutionary Programming—Proceedings of the Third Annual Conference, World Scientific Publishing, San Diego, CA, USA, pp 233–241, URL http://dynamics.org/~altenber/PAPERS/EPIGP/
4. Alves da Silva AP, Abrao PJ (2002) Applications of evolutionary computation in electric power systems. In: Fogel DB, El-Sharkawi MA, Yao X, Greenwood G, Iba H, Marrow P, Shackleton M (eds) Proceedings of the 2002 Congress on Evolutionary Computation CEC2002, IEEE Press, pp 1057–1062, DOI doi:10.1109/CEC.2002.1004389
5. Ando D, Dahlsted P, Nordahl M, Iba H (2007) Interactive GP with tree representation of classical music pieces. In: Giacobini M, Brabazon A, Cagnoni S, Di Caro GA, Drechsler R, Farooq M, Fink A, Lutton E, Machado P, Minner S, O'Neill M, Romero J, Rothlauf F, Squillero G, Takagi H, Uyar AS, Yang S (eds) Applications of Evolutionary Computing, EvoWorkshops 2007: EvoCOMNET, EvoFIN, EvoIASP, EvoInteraction, EvoMUSART, EvoSTOC, EvoTransLog, Springer Verlag, Valencia, Spain, LNCS, vol 4448, pp 577–584, DOI doi:10.1007/978-3-540-71805-5_63
6. Andre D, Koza JR (1996) Parallel genetic programming: A scalable implementation using the transputer network architecture. In: Angeline PJ, Kinnear, Jr KE (eds) Advances in Genetic Programming 2, MIT Press, Cambridge, MA, USA, chap 16, pp 317–338
7. Andre D, Koza JR (1998) A parallel implementation of genetic programming that achieves super-linear performance. Information Sciences 106(3–4):201–218, URL http://www.sciencedirect.com/science/article/B6V0C-3TKS65B-21/2/22b9842f820b08883990bbae1d889c03

8. Andre D, Bennett III FH, 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, Goldberg DE, Fogel DB, Riolo RL (eds) Genetic Programming 1996: Proceedings of the First Annual Conference, MIT Press, Stanford University, CA, USA, pp 3–11, URL http://www.genetic-programming.com/jkpdf/gp1996gkl.pdf
9. Angeline PJ (1996) An investigation into the sensitivity of genetic programming to the frequency of leaf selection during subtree crossover. In: Koza JR, Goldberg DE, Fogel DB, Riolo RL (eds) Genetic Programming 1996: Proceedings of the First Annual Conference, MIT Press, Stanford University, CA, USA, pp 21–29, URL http://www.natural-selection.com/Library/1996/gp96.zip
10. Angeline PJ (1997) Subtree crossover: Building block engine or macromutation? In: Koza JR, Deb K, Dorigo M, Fogel DB, Garzon M, Iba H, Riolo RL (eds) Genetic Programming 1997: Proceedings of the Second Annual Conference, Morgan Kaufmann, Stanford University, CA, USA, pp 9–17
11. Angeline PJ (1998) Multiple interacting programs: A representation for evolving complex behaviors. Cybernetics and Systems 29(8):779–806, URL http://www.natural-selection.com/Library/1998/mips3.pdf
12. Angeline PJ, Kinnear, Jr KE (eds) (1996) Advances in Genetic Programming 2. MIT Press, Cambridge, MA, USA, URL http://www.cs.bham.ac.uk/~wbl/aigp2.html
13. Angeline PJ, Pollack JB (1992) The evolutionary induction of subroutines. In: Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society, Lawrence Erlbaum, Bloomington, Indiana, USA, pp 236–241, URL http://www.demo.cs.brandeis.edu/papers/glib92.pdf
14. Arkov V, Evans C, Fleming PJ, Hill DC, Norton JP, Pratt I, Rees D, Rodriguez-Vazquez K (2000) System identification strategies applied to aircraft gas turbine engines. Annual Reviews in Control 24(1):67–81, URL http://www.sciencedirect.com/science/article/B6V0H-482MDPD-8/2/dd470648e2228c84efe7e14ca3841b7e
15. Austin MP, Bates G, Dempster MAH, Leemans V, Williams SN (2004) Adaptive systems for foreign exchange trading. Quantitative Finance 4(4):37–45, DOI doi:10.1080/14697680400008593, URL http://www-cfr.jbs.cam.ac.uk/archive/PRESENTATIONS/seminars/2006/dempster2.pdf
16. Azaria Y, Sipper M (2005a) GP-gammon: Genetically programming backgammon players. Genetic Programming and Evolvable Machines 6(3):283–300, DOI doi:10.1007/s10710-005-2990-0, URL http://www.cs.bgu.ac.il/~sipper/papabs/gpgammon.pdf, published online: 12 August 2005
17. Azaria Y, Sipper M (2005b) Using GP-gammon: Using genetic programming to evolve backgammon players. In: Keijzer M, Tettamanzi A, Collet P, van Hemert JI, Tomassini M (eds) Proceedings of the 8th European Conference on Genetic Programming, Springer, Lausanne, Switzerland, Lecture Notes in Computer Science, vol 3447, pp 132–142, URL http://springerlink.metapress.com/openurl.asp?genre=article&issn=0302-9743&volume=3447&spage=132
18. Babovic V (1996) Emergence, evolution, intelligence; Hydroinformatics - A study of distributed and decentralised computing using intelligent agents. A. A. Balkema Publishers, Rotterdam, Holland
19. Bader-El-Den M, Poli R (2007a) Generating sat local-search heuristics using a gp hyper-heuristic framework. In: Proceedings of Evolution Artificielle

20. Bader-El-Den MB, Poli R (2007b) A GP-based hyper-heuristic framework for evolving 3-SAT heuristics. In: Thierens D, Beyer HG, Bongard J, Branke J, Clark JA, Cliff D, Congdon CB, Deb K, Doerr B, Kovacs T, Kumar S, Miller JF, Moore J, Neumann F, Pelikan M, Poli R, Sastry K, Stanley KO, Stutzle T, Watson RA, Wegener I (eds) GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation, ACM Press, London, vol 2, pp 1749–1749, URL http://www.cs.bham.ac.uk/~wbl/biblio/gecco2007/docs/p1749.pdf
21. Bains W, Gilbert R, Sviridenko L, Gascon JM, Scoffin R, Birchall K, Harvey I, Caldwell J (2002) Evolutionary computational methods to predict oral bioavailability QSPRs. Current Opinion in Drug Discovery and Development 5(1):44–51
22. Baker JE (1987) Reducing bias and inefficiency in the selection algorithm. In: Grefenstette JJ (ed) Proceedings of the Second International Conference on Genetic Algorithms and their Application, Lawrence Erlbaum Associates, Cambridge, MA, USA, pp 14–21
23. Balic J (1999) Flexible Manufacturing Systems; Development - Structure - Operation - Handling - Tooling. Manufacturing technology, DAAAM International, Vienna
24. Banzhaf W (1993) Genetic programming for pedestrians. In: Forrest S (ed) Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93, Morgan Kaufmann, University of Illinois at Urbana-Champaign, p 628, URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/ftp.io.com/papers/GenProg_forPed.ps.Z
25. 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, CA, USA
26. Barrett SJ (2003) Recurring analytical problems within drug discovery and development. In: Scheffer T, Leser U (eds) Data Mining and Text Mining for Bioinformatics: Proceedings of the European Workshop, Dubrovnik, Croatia, pp 6–7, URL http://www2.informatik.hu-berlin.de/~scheffer/publications/ProceedingsWS2003.pdf, invited talk
27. Barrett SJ, Langdon WB (2006) Advances in the application of machine learning techniques in drug discovery, design and development. In: Tiwari A, Knowles J, Avineri E, Dahal K, Roy R (eds) Applications of Soft Computing: Recent Trends, Springer, On the World Wide Web, Advances in Soft Computing, pp 99–110, URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/barrett_2005_WSC.pdf
28. Bennett III FH (1996) Automatic creation of an efficient multi-agent architecture using genetic programming with architecture-altering operations. In: Koza JR, Goldberg DE, Fogel DB, Riolo RL (eds) Genetic Programming 1996: Proceedings of the First Annual Conference, MIT Press, Stanford University, CA, USA, pp 30–38, URL http://cognet.mit.edu/library/books/view?isbn=0262611279
29. Bennett III FH, Koza JR, Shipman J, Stiffelman O (1999) Building a parallel computer system for $18,000 that performs a half peta-flop per day. In: Banzhaf W, Daida J, Eiben AE, Garzon MH, Honavar V, Jakiela M, Smith RE (eds) Proceedings of the Genetic and Evolutionary Computation Conference, Morgan Kaufmann, Orlando, Florida, USA, vol 2, pp 1484–1490, URL http://www.genetic-programming.com/jkpdf/gecco1999beowulf.pdf

30. Bhanu B, Lin Y, Krawiec K (2005) Evolutionary Synthesis of Pattern Recognition Systems. Monographs in Computer Science, Springer-Verlag, New York, URL http://www.springer.com/west/home/computer/imaging?SGWID=4-14%9-22-39144807-detailsPage=ppmmedia—aboutThisBook
31. Blickle T (1996) Theory of evolutionary algorithms and application to system synthesis. PhD thesis, Swiss Federal Institute of Technology, Zurich, URL http://www.handshake.de/user/blickle/publications/diss.pdf
32. Brabazon A, O'Neill M (2006) Biologically Inspired Algorithms for Financial Modeling. Natural Computing Series, Springer
33. Brameier M, Banzhaf W (2001) A comparison of linear genetic programming and neural networks in medical data mining. IEEE Transactions on Evolutionary Computation 5(1):17–26, URL http://web.cs.mun.ca/˜banzhaf/papers/ieee_taec.pdf
34. Brameier M, Banzhaf W (2007) Linear Genetic Programming. No. XVI in Genetic and Evolutionary Computation, Springer, URL http://www.springer.com/west/home/default?SGWID=4-40356-22-173660820-0
35. Brameier M, Haan J, Krings A, MacCallum RM (2006) Automatic discovery of cross-family sequence features associated with protein function. BMC bioinformatics [electronic resource] 7(16), DOI doi:10.1186/1471-2105-7-16, URL http://www.biomedcentral.com/content/pdf/1471-2105-7-16.pdf
36. Brave S (1996) Evolving recursive programs for tree search. In: Angeline PJ, Kinnear, Jr KE (eds) Advances in Genetic Programming 2, MIT Press, Cambridge, MA, USA, chap 10, pp 203–220
37. Brezocnik M (2000) Uporaba genetskega programiranja v inteligentnih proizvodnih sistemih. University of Maribor, Faculty of mechanical engineering, Maribor, Slovenia, URL http://maja.uni-mb.si/slo/Knjige/2000-03-mon/index.htm
38. Brezocnik M, Balic J, Gusel L (2000) Artificial intelligence approach to determination of flow curve. Journal for technology of plasticity 25(1–2):1–7
39. Buason G, Bergfeldt N, Ziemke T (2005) Brains, bodies, and beyond: Competitive co-evolution of robot controllers, morphologies and environments. Genetic Programming and Evolvable Machines 6(1):25–51, DOI doi:10.1007/s10710-005-7618-x
40. Burke E, Kendall G, Newall J, Hart E, Ross P, Schulenburg S (2003) Hyper-heuristics: an emerging direction in modern search technology. In: Glover F, Kochenberger G (eds) Handbook of Metaheuristics, Kluwer Academic Publishers, pp 457–474
41. Burke EK, Hyde MR, Kendall G (2006) Evolving bin packing heuristics with genetic programming. In: Runarsson TP, Beyer HG, Burke E, Merelo-Guervos JJ, Whitley LD, Yao X (eds) Parallel Problem Solving from Nature - PPSN IX, Springer-Verlag, Reykjavik, Iceland, LNCS, vol 4193, pp 860–869, DOI doi:10.1007/11844297_87, URL http://www.cs.nott.ac.uk/˜mvh/ppsn2006.pdf
42. Burke EK, Hyde MR, Kendall G, Woodward J (2007) Automatic heuristic generation with genetic programming: evolving a jack-of-all-trades or a master of one. In: Thierens D, Beyer HG, Bongard J, Branke J, Clark JA, Cliff D, Congdon CB, Deb K, Doerr B, Kovacs T, Kumar S, Miller JF, Moore J, Neumann F, Pelikan M, Poli R, Sastry K, Stanley KO, Stutzle T, Watson RA, Wegener I (eds) GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation, ACM Press, London, vol 2, pp 1559–1565, URL http://www.cs.bham.ac.uk/ wbl/biblio/gecco2007/docs/p1559.pdf

43. Buxton BF, Langdon WB, Barrett SJ (2001) Data fusion by intelligent classifier combination. Measurement and Control 34(8):229–234, URL http://www.cs.ucl.ac.uk/staff/W.Langdon/mc/
44. Cagnoni S, Bergenti F, Mordonini M, Adorni G (2005) Evolving binary classifiers through parallel computation of multiple fitness cases. IEEE Transactions on Systems, Man and Cybernetics - Part B 35(3):548–555, DOI doi:10.1109/TSMCB.2005.846671
45. Cai W, Pacheco-Vega A, Sen M, Yang KT (2006) Heat transfer correlations by symbolic regression. International Journal of Heat and Mass Transfer 49(23-24):4352–4359, DOI doi:10.1016/j.ijheatmasstransfer.2006.04.029
46. Castillo F, Kordon A, Smits G (2006) Robust pareto front genetic programming parameter selection based on design of experiments and industrial data. In: Riolo RL, Soule T, Worzel B (eds) Genetic Programming Theory and Practice IV, Genetic and Evolutionary Computation, vol 5, Springer, Ann Arbor
47. Chami M, Robilliard D (2002) Inversion of oceanic constituents in case I and II waters with genetic programming algorithms. Applied Optics 41(30):6260–6275, URL http://ao.osa.org/ViewMedia.cfm?id=70258&seq=0
48. Channon A (2006) Unbounded evolutionary dynamics in a system of agents that actively process and transform their environment. Genetic Programming and Evolvable Machines 7(3):253–281, DOI doi:10.1007/s10710-006-9009-3
49. Chao DL, Forrest S (2003) Information immune systems. Genetic Programming and Evolvable Machines 4(4):311–331, DOI doi:10.1023/A:1026139027539
50. Cheang SM, Leung KS, Lee KH (2006) Genetic parallel programming: Design and implementation. Evolutionary Computation 14(2):129–156, DOI doi:10.1162/evco.2006.14.2.129
51. Chen SH (ed) (2002) Genetic Algorithms and Genetic Programming in Computational Finance. Kluwer Academic Publishers, Dordrecht, URL http://www.springer.com/west/home/business?SGWID=4-40517-22-3%3195998-detailsPage=ppmmedia|toc
52. Chen SH, Liao CC (2005) Agent-based computational modeling of the stock price-volume relation. Information Sciences 170(1):75–100, DOI doi:10.1016/j.ins.2003.03.026, URL http://www.sciencedirect.com/science/article/B6V0C-4B3JHTS-6/2/9e023835b1c70f176d1903dd3a8b638e
53. Chen SH, Wang HS, Zhang BT (1999) Forecasting high-frequency financial time series with evolutionary neural trees: The case of heng-sheng stock index. In: Arabnia HR (ed) Proceedings of the International Conference on Artificial Intelligence, IC-AI '99, CSREA Press, Las Vegas, Nevada, USA, vol 2, pp 437–443, URL http://bi.snu.ac.kr/Publications/Conferences/International/ICAI99.ps
54. Chen SH, Duffy J, Yeh CH (2002) Equilibrium selection via adaptation: Using genetic programming to model learning in a coordination game. The Electronic Journal of Evolutionary Modeling and Economic Dynamics
55. Chitty DM (2007) A data parallel approach to genetic programming using programmable graphics hardware. In: Thierens D, Beyer HG, Bongard J, Branke J, Clark JA, Cliff D, Congdon CB, Deb K, Doerr B, Kovacs T, Kumar S, Miller JF, Moore J, Neumann F, Pelikan M, Poli R, Sastry K, Stanley KO, Stutzle T, Watson RA, Wegener I (eds) GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation, ACM Press, London, vol 2, pp 1566–1573, URL http://www.cs.bham.ac.uk/~wbl/biblio/gecco2007/docs/p1566.pdf

56. Chong FS, Langdon WB (1999) Java based distributed genetic programming on the internet. In: Banzhaf W, Daida J, Eiben AE, Garzon MH, Honavar V, Jakiela M, Smith RE (eds) Proceedings of the Genetic and Evolutionary Computation Conference, Morgan Kaufmann, Orlando, Florida, USA, vol 2, p 1229, URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/p.chong/DGPposter.pdf, full text in technical report CSRP-99-7
57. Ciesielski V, Li X (2004) Analysis of genetic programming runs. In: Mckay RI, Cho SB (eds) Proceedings of The Second Asian-Pacific Workshop on Genetic Programming, Cairns, Australia, URL http://goanna.cs.rmit.edu.au/~xiali/pub/ai04.vc.pdf
58. Cilibrasi R, Vitanyi PMB (2005) Clustering by compression. IEEE Transactions on Information Theory 51(4):1523–1545, URL http://homepages.cwi.nl/~paulv/papers/cluster.pdf
59. Cilibrasi R, Vitanyi P, de Wolf R (2004) Algorithmic clustering of music based on string compression. Computer Music Journal 28(4):49–67, URL http://homepages.cwi.nl/~paulv/papers/music.pdf
60. Collins RJ (1992) Studies in artificial evolution. PhD thesis, UCLA, Artificial Life Laboratory, Department of Computer Science, University of California, Los Angeles, LA CA 90024, USA
61. Corno F, Sanchez E, Squillero G (2005) Evolving assembly programs: how games help microprocessor validation. Evolutionary Computation, IEEE Transactions on 9(6):695–706
62. Costelloe D, Ryan C (2007) Towards models of user preferences in interactive musical evolution. In: Thierens D, Beyer HG, Bongard J, Branke J, Clark JA, Cliff D, Congdon CB, Deb K, Doerr B, Kovacs T, Kumar S, Miller JF, Moore J, Neumann F, Pelikan M, Poli R, Sastry K, Stanley KO, Stutzle T, Watson RA, Wegener I (eds) GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation, ACM Press, London, vol 2, pp 2254–2254, URL http://www.cs.bham.ac.uk/~wbl/biblio/gecco2007/docs/p2254.pdf
63. Cranmer K, Bowman RS (2005) PhysicsGP: A genetic programming approach to event selection. Computer Physics Communications 167(3):165–176, DOI doi:10.1016/j.cpc.2004.12.006
64. Crawford-Marks R, Spector L (2002) Size control via size fair genetic operators in the PushGP genetic programming system. In: Langdon WB, Cantú-Paz E, Mathias K, Roy R, Davis D, Poli R, Balakrishnan K, Honavar V, Rudolph G, Wegener J, Bull L, Potter MA, Schultz AC, Miller JF, Burke E, Jonoska N (eds) GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, Morgan Kaufmann Publishers, New York, pp 733–739, URL http://alum.hampshire.edu/~rpc01/gp234.pdf
65. Crepeau RL (1995) Genetic evolution of machine language software. In: Rosca JP (ed) Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications, Tahoe City, California, USA, pp 121–134, URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/GEMS_Article.pdf
66. Curry R, Lichodzijewski P, Heywood MI (2007) Scaling genetic programming to large datasets using hierarchical dynamic subset selection. IEEE Transactions on Systems, Man, and Cybernetics: Part B - Cybernetics 37(4):1065–1073, DOI doi:10.1109/TSMCB.2007.896406, URL http://www.cs.dal.ca/~mheywood/X-files/GradPubs.html#curry

67. Daida JM, Hommes JD, Bersano-Begey TF, Ross SJ, Vesecky JF (1996) Algorithm discovery using the genetic programming paradigm: Extracting low-contrast curvilinear features from SAR images of arctic ice. In: Angeline PJ, Kinnear, Jr KE (eds) Advances in Genetic Programming 2, MIT Press, Cambridge, MA, USA, chap 21, pp 417–442, URL http://sitemaker.umich.edu/daida/files/GP2_cha21.pdf
68. Dassau E, Grosman B, Lewin DR (2006) Modeling and temperature control of rapid thermal processing. Computers and Chemical Engineering 30(4):686–697, DOI doi:10.1016/j.compchemeng.2005.11.007, URL http://tx.technion.ac.il/~dlewin/publications/rtp_paper_v9.pdf
69. Davis TE, Principe JC (1993) A Markov chain framework for the simple genetic algorithm. Evolutionary Computation 1(3):269–288
70. Day JP, Kell DB, Griffith GW (2002) Differentiation of phytophthora infestans sporangia from other airborne biological particles by flow cytometry. Applied and Environmental Microbiology 68(1):37–45, DOI doi:10.1128/AEM.68.1.37-45.2002, URL http://intl-aem.asm.org/cgi/reprint/68/1/37.pdf
71. de Sousa JS, de CT Gomes L, Bezerra GB, de Castro LN, Von Zuben FJ (2004) An immune-evolutionary algorithm for multiple rearrangements of gene expression data. Genetic Programming and Evolvable Machines 5(2):157–179, DOI doi:10.1023/B:GENP.0000023686.59617.57
72. De Stefano C, Cioppa AD, Marcelli A (2002) Character preclassification based on genetic programming. Pattern Recognition Letters 23(12):1439–1448, DOI doi:10.1016/S0167-8655(02)00104-6, URL http://www.sciencedirect.com/science/article/B6V15-45J91MV-4/2/3e5c2ac0c51428d0f7ea9fc0142f6790
73. Deb K (2001) Multi-objective optimization using evolutionary algorithms. Wiley
74. Dempster MAH, Jones CM (2000) A real-time adaptive trading system using genetic programming. Quantitative Finance 1:397–413, URL http://mahd-pc.jbs.cam.ac.uk/archive/PAPERS/2000/geneticprogramming.pdf
75. Dempster MAH, Payne TW, Romahi Y, Thompson GWP (2001) Computational learning techniques for intraday FX trading using popular technical indicators. IEEE Transactions on Neural Networks 12(4):744–754, DOI doi:10.1109/72.935088, URL http://mahd-pc.jbs.cam.ac.uk/archive/PAPERS/2000/ieeetrading.pdf
76. Deschaine L (2006) Using information fusion, machine learning, and global optimisation to increase the accuracy of finding and understanding items interest in the subsurface. GeoDrilling International (122):30–32, URL http://www.mining-journal.com/gdi_magazine/pdf/GDI0605scr.pdf
77. Deschaine LM, Patel JJ, Guthrie RD, Grimski JT, Ades MJ (2001) Using linear genetic programming to develop a C/C++ simulation model of a waste incinerator. In: Ades M (ed) Advanced Technology Simulation Conference, Seattle, URL http://www.aimlearning.com/Environmental.Engineering.pdf
78. Deschaine LM, Hoover RA, Skibinski JN, Patel JJ, Francone F, Nordin P, Ades MJ (2002) Using machine learning to compliment and extend the accuracy of UXO discrimination beyond the best reported results of the jefferson proving ground technology demonstration. In: 2002 Advanced Technology Simulation Conference, San Diego, CA, USA, URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/deschaine/ASTC_2002_UXO Finder_Invention_Paper.pdf

79. D'haeseleer P (1994) Context preserving crossover in genetic programming. In: Proceedings of the 1994 IEEE World Congress on Computational Intelligence, IEEE Press, Orlando, Florida, USA, vol 1, pp 256–261, URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/ftp.io.com/papers/WCCI94_CPC.ps.Z
80. D'haeseleer P, Bluming J (1994) Effects of locality in individual and population evolution. In: Kinnear, Jr KE (ed) Advances in Genetic Programming, MIT Press, chap 8, pp 177–198, URL http://cognet.mit.edu/library/books/view?isbn=0262111888
81. 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, Beyer HG, Bongard J, Branke J, Clark JA, Cliff D, Congdon CB, Deb K, Doerr B, Kovacs T, Kumar S, Miller JF, Moore J, Neumann F, Pelikan M, Poli R, Sastry K, Stanley KO, Stutzle T, Watson RA, Wegener I (eds) GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation, ACM Press, London, vol 2, pp 1588–1595, URL http://www.cs.bham.ac.uk/~wbl/biblio/gecco2007/docs/p1588.pdf
82. Dolinsky JU, Jenkinson ID, Colquhoun GJ (2007) Application of genetic programming to the calibration of industrial robots. Computers in Industry 58(3):255–264, DOI doi:10.1016/j.compind.2006.06.003
83. Domingos RP, Schirru R, Martinez AS (2005) Soft computing systems applied to PWR's xenon. Progress in Nuclear Energy 46(3–4):297–308, DOI doi:10.1016/j.pnucene.2005.03.011
84. Dracopoulos DC (1997) Evolutionary Learning Algorithms for Neural Adaptive Control. Perspectives in Neural Computing, Springer Verlag, P.O. Box 31 13 40, D-10643 Berlin, Germany, URL http://www.springer.de/catalog/html-files/deutsch/comp/3540761616.html
85. Droste S, Jansen T, Rudolph G, Schwefel HP, Tinnefeld K, Wegener I (2003) Theory of evolutionary algorithms and genetic programming. In: Schwefel HP, Wegener I, Weinert K (eds) Advances in Computational Intelligence: Theory and Practice, Natural Computing Series, Springer, chap 5, pp 107–144
86. Ebner M, Reinhardt M, Albert J (2005) Evolution of vertex and pixel shaders. In: Keijzer M, Tettamanzi A, Collet P, van Hemert JI, Tomassini M (eds) Proceedings of the 8th European Conference on Genetic Programming, Springer, Lausanne, Switzerland, Lecture Notes in Computer Science, vol 3447, pp 261–270, DOI doi:10.1007/b107383, URL http://springerlink.metapress.com/openurl.asp?genre=article&issn=0302-9743&volume=3447&spage=261
87. Eiben AE, Smith JE (2003) Introduction to Evolutionary Computing. Springer, URL http://www.cs.vu.nl/~gusz/ecbook/ecbook.html
88. Eklund SE (2002) A massively parallel GP engine in VLSI. In: Fogel DB, El-Sharkawi MA, Yao X, Greenwood G, Iba H, Marrow P, Shackleton M (eds) Proceedings of the 2002 Congress on Evolutionary Computation CEC2002, IEEE Press, pp 629–633
89. Ellis DI, Broadhurst D, Kell DB, Rowland JJ, Goodacre R (2002) Rapid and quantitative detection of the microbial spoilage of meat by fourier transform infrared spectroscopy and machine learning. Applied and Environmental Microbiology 68(6):2822–2828, DOI doi:10.1128/AEM.68.6.2822?2828.2002, URL http://dbkgroup.org/Papers/app_%20env_microbiol_68_(2822).pdf
90. Ellis DI, Broadhurst D, Goodacre R (2004) Rapid and quantitative detection of the microbial spoilage of beef by fourier transform infrared spectroscopy and

machine learning. Analytica Chimica Acta 514(2):193–201, DOI doi:10.1016/j.aca.2004.03.060, URL http://dbkgroup.org/dave_files/ACAbeef04.pdf
91. Eriksson R, Olsson B (2004) Adapting genetic regulatory models by genetic programming. Biosystems 76(1–3):217–227, DOI doi:10.1016/j.biosystems.2004.05.014, URL http://www.sciencedirect.com/science/article/B6T2K-4D09KY2-7/2/1abfe196bb4afc60afc3311cadb75d66
92. Esparcia-Alcazar AI, Sharman KC (1996) Genetic programming techniques that evolve recurrent neural networks architectures for signal processing. In: IEEE Workshop on Neural Networks for Signal Processing, Seiko, Kyoto, Japan
93. Evans C, Fleming PJ, Hill DC, Norton JP, Pratt I, Rees D, Rodriguez-Vazquez K (2001) Application of system identification techniques to aircraft gas turbine engines. Control Engineering Practice 9(2):135–148, URL http://www.sciencedirect.com/science/article/B6V2H-4280YP2-3/1/24d44180070f91dea854032d98f9187a
94. Federman F, Sparkman G, Watt S (1999) Representation of music in a learning classifier system utilizing bach chorales. In: Banzhaf W, Daida J, Eiben AE, Garzon MH, Honavar V, Jakiela M, Smith RE (eds) Proceedings of the Genetic and Evolutionary Computation Conference, Morgan Kaufmann, Orlando, Florida, USA, vol 1, p 785
95. Felton MJ (2000) Survival of the fittest in drug design. Modern Drug Discovery 3(9):49–50, URL http://pubs.acs.org/subscribe/journals/mdd/v03/i09/html/felton.html
96. Fernandez F, Sanchez JM, Tomassini M, Gomez JA (1999) A parallel genetic programming tool based on PVM. In: Dongarra J, Luque E, Margalef T (eds) Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proceedings of the 6th European PVM/MPI Users' Group Meeting, Springer-Verlag, Barcelona, Spain, Lecture Notes in Computer Science, vol 1697, pp 241–248
97. Fernandez F, Tomassini M, Vanneschi L (2003) An empirical study of multipopulation genetic programming. Genetic Programming and Evolvable Machines 4(1):21–51, DOI doi:10.1023/A:1021873026259
98. Folino G, Pizzuti C, Spezzano G (2003) A scalable cellular implementation of parallel genetic programming. IEEE Transactions on Evolutionary Computation 7(1):37–53
99. Foster JA (2001) Review: Discipulus: A commercial genetic programming system. Genetic Programming and Evolvable Machines 2(2):201–203, DOI doi:10.1023/A:1011516717456
100. Francone FD, Deschaine LM (2004) Getting it right at the very start – building project models where data is expensive by combining human expertise, machine learning and information theory. In: 2004 Business and Industry Symposium, Washington, DC, URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/deschaine/ASTC_2004_Getting_It_Right_from_the_Very_Start.pdf
101. Francone FD, Conrads M, Banzhaf W, Nordin P (1999) Homologous crossover in genetic programming. In: Banzhaf W, Daida J, Eiben AE, Garzon MH, Honavar V, Jakiela M, Smith RE (eds) Proceedings of the Genetic and Evolutionary Computation Conference, Morgan Kaufmann, Orlando, Florida, USA, vol 2, pp 1021–1026, URL http://www.cs.bham.ac.uk/~wbl/biblio/gecco1999/GP-463.pdf

102. Francone FD, Deschaine LM, Warren JJ (2007) Discrimination of munitions and explosives of concern at F.E. warren AFB using linear genetic programming. In: Thierens D, Beyer HG, Bongard J, Branke J, Clark JA, Cliff D, Congdon CB, Deb K, Doerr B, Kovacs T, Kumar S, Miller JF, Moore J, Neumann F, Pelikan M, Poli R, Sastry K, Stanley KO, Stutzle T, Watson RA, Wegener I (eds) GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation, ACM Press, London, vol 2, pp 1999–2006, URL http://www.cs.bham.ac.uk/~wbl/biblio/gecco2007/docs/p1999.pdf
103. Fukunaga A (2002) Automated discovery of composite SAT variable selection heuristics. In: Proceedings of the National Conference on Artificial Intelligence (AAAI), pp 641–648
104. Fukunaga AS (2004) Evolving local search heuristics for SAT using genetic programming. In: Deb K, Poli R, Banzhaf W, Beyer HG, Burke E, Darwen P, Dasgupta D, Floreano D, Foster J, Harman M, Holland O, Lanzi PL, Spector L, Tettamanzi A, Thierens D, Tyrrell A (eds) Genetic and Evolutionary Computation – GECCO-2004, Part II, Springer-Verlag, Seattle, WA, USA, Lecture Notes in Computer Science, vol 3103, pp 483–494, DOI doi:10.1007/b98645, URL http://alexf04.maclisp.org/gecco2004.pdf
105. Funes P, Sklar E, Juille H, Pollack J (1998a) Animal-animat coevolution: Using the animal population as fitness function. In: Pfeifer R, Blumberg B, Meyer JA, Wilson SW (eds) From Animals to Animats 5: Proceedings of the Fifth International Conference on Simulation of Adaptive Behavior., MIT Press, Zurich, Switzerland, pp 525–533, URL http://www.demo.cs.brandeis.edu/papers/tronsab98.html
106. Funes P, Sklar E, Juille H, Pollack J (1998b) Animal-animat coevolution: Using the animal population as fitness function. In: Pfeifer R, Blumberg B, Meyer JA, Wilson SW (eds) From Animals to Animats 5: Proceedings of the Fifth International Conference on Simulation of Adaptive Behavior, MIT Press, Zurich, Switzerland, pp 525–533, URL http://www.demo.cs.brandeis.edu/papers/tronsab98.pdf
107. Gagne C, Parizeau M (2006) Genetic engineering of hierarchical fuzzy regional representations for handwritten character recognition. International Journal on Document Analysis and Recognition 8(4):223–231, DOI doi:10.1007/s10032-005-0005-6, URL http://vision.gel.ulaval.ca/fr/publications/Id_607/PublDetails.php
108. Gagné C, Parizeau M (2007) Co-evolution of nearest neighbor classifiers. International Journal of Pattern Recognition and Artificial Intelligence 21(5):921–946, DOI doi:10.1142/S0218001407005752, URL http://vision.gel.ulaval.ca/en/publications/Id_692/PublDetails.php
109. Garcia-Almanza AL, Tsang EPK (2006) Forecasting stock prices using genetic programming and chance discovery. In: 12th International Conference On Computing In Economics And Finance, p number 489, URL http://repec.org/sce2006/up.13879.1141401469.pdf
110. Gathercole C, Ross P (1994) Dynamic training subset selection for supervised learning in genetic programming. In: Davidor Y, Schwefel HP, Männer R (eds) Parallel Problem Solving from Nature III, Springer-Verlag, Jerusalem, LNCS, vol 866, pp 312–321, URL http://citeseer.ist.psu.edu/gathercole94dynamic.html
111. Gathercole C, Ross P (1997) Tackling the boolean even N parity problem with genetic programming and limited-error fitness. In: Koza JR, Deb K, Dorigo

M, Fogel DB, Garzon M, Iba H, Riolo RL (eds) Genetic Programming 1997: Proceedings of the Second Annual Conference, Morgan Kaufmann, Stanford University, CA, USA, pp 119–127, URL http://citeseer.ist.psu.edu/79389.html
112. Gelly S, Teytaud O, Bredeche N, Schoenauer M (2006) Universal consistency and bloat in GP. Revue d'Intelligence Artificielle 20(6):805–827, URL http://hal.inria.fr/docs/00/11/28/40/PDF/riabloat.pdf, issue on New Methods in Machine Learning. Theory and Applications
113. Gilbert RJ, Goodacre R, Woodward AM, Kell DB (1997) Genetic programming: A novel method for the quantitative analysis of pyrolysis mass spectral data. ANALYTICAL CHEMISTRY 69(21):4381–4389, DOI doi:10.1021/ac970460j, URL http://pubs.acs.org/journals/ancham/article.cgi/ancham/1997/69/i21/pdf/ac970460j.pdf
114. Globus A, Lawton J, Wipke T (1998) Automatic molecular design using evolutionary techniques. In: Globus A, Srivastava D (eds) The Sixth Foresight Conference on Molecular Nanotechnology, Westin Hotel in Santa Clara, CA, USA, URL http://www.foresight.org/Conferences/MNT6/Papers/Globus/index.html
115. Goldberg DE (1989) Genetic Algorithms in Search Optimization and Machine Learning. Addison-Wesley
116. Goldberg DE, Kargupta H, Horn J, Cantu-Paz E (1995) Critical deme size for serial and parallel genetic algorithms. Tech. rep., Illinois Genetic Algorithms Laboratory, Department of General Engineering, University of Illinois at Urbana-Champaign, Il 61801, USA, illiGAL Report no 95002
117. Goodacre R (2003) Explanatory analysis of spectroscopic data using machine learning of simple, interpretable rules. Vibrational Spectroscopy 32(1):33–45, DOI doi:10.1016/S0924-2031(03)00045-6, URL http://www.biospec.net/learning/Metab06/Goodacre-FTIRmaps.pdf, a collection of Papers Presented at Shedding New Light on Disease: Optical Diagnostics for the New Millennium (SPEC 2002) Reims, France 23–27 June 2002
118. Goodacre R, Gilbert RJ (1999) The detection of caffeine in a variety of beverages using curie-point pyrolysis mass spectrometry and genetic programming. The Analyst 124:1069–1074
119. Goodacre R, Shann B, Gilbert RJ, Timmins EM, McGovern AC, Alsberg BK, Kell DB, Logan NA (2000) The detection of the dipicolinic acid biomarker in bacillus spores using curie-point pyrolysis mass spectrometry and fourier-transform infrared spectroscopy. Analytical Chemistry 72(1):119–127, DOI doi:10.1021/ac990661i, URL http://pubs.acs.org/cgi-bin/article.cgi/ancham/2000/72/i01/html/ac990661i.html
120. Goodacre R, Vaidyanathan S, Dunn WB, Harrigan GG, Kell DB (2004) Metabolomics by numbers: acquiring and understanding global metabolite data. Trends in Biotechnology 22(5):245–252, DOI doi:10.1016/j.tibtech.2004.03.007, URL http://dbkgroup.org/Papers/trends%20in%20biotechnology_22_(24%5).pdf
121. Gruau F (1994a) Genetic micro programming of neural networks. In: Kinnear, Jr KE (ed) Advances in Genetic Programming, MIT Press, chap 24, pp 495–518, URL http://cognet.mit.edu/library/books/view?isbn=0262111888
122. Gruau F (1994b) Neural network synthesis using cellular encoding and the genetic algorithm. PhD thesis, Laboratoire de l'Informatique du Parallilisme, Ecole Normale Supirieure de Lyon, France, URL ftp://ftp.ens-lyon.fr/pub/LIP/Rapports/PhD/PhD1994/PhD1994-01-E.ps.Z

123. Gruau F (1996) On using syntactic constraints with genetic programming. In: Angeline PJ, Kinnear, Jr KE (eds) Advances in Genetic Programming 2, MIT Press, Cambridge, MA, USA, chap 19, pp 377–394
124. Gruau F, Whitley D (1993) Adding learning to the cellular development process: a comparative study. Evolutionary Computation 1(3):213–233
125. Gustafson S, Burke EK (2006) The speciating island model: An alternative parallel evolutionary algorithm. Journal of Parallel and Distributed Computing 66(8):1025–1036, DOI doi:10.1016/j.jpdc.2006.04.017, parallel Bioinspired Algorithms
126. Gustafson S, Burke EK, Krasnogor N (2005) On improving genetic programming for symbolic regression. In: Corne D, Michalewicz Z, Dorigo M, Eiben G, Fogel D, Fonseca C, Greenwood G, Chen TK, Raidl G, Zalzala A, Lucas S, Paechter B, Willies J, Guervos JJM, Eberbach E, McKay B, Channon A, Tiwari A, Volkert LG, Ashlock D, Schoenauer M (eds) Proceedings of the 2005 IEEE Congress on Evolutionary Computation, IEEE Press, Edinburgh, UK, vol 1, pp 912–919
127. Hampo RJ, Marko KA (1992) Application of genetic programming to control of vehicle systems. In: Proceedings of the Intelligent Vehicles '92 Symposium, june 29 July 1, 1992, Detroit, Mi, USA
128. Handley S (1993) Automatic learning of a detector for alpha-helices in protein sequences via genetic programming. In: Forrest S (ed) Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93, Morgan Kaufmann, University of Illinois at Urbana-Champaign, pp 271–278
129. Handley S (1994) On the use of a directed acyclic graph to represent a population of computer programs. In: Proceedings of the 1994 IEEE World Congress on Computational Intelligence, IEEE Press, Orlando, Florida, USA, vol 1, pp 154–159, DOI doi:10.1109/ICEC.1994.350024
130. Harding S, Banzhaf W (2007) Fast genetic programming on GPUs. In: Ebner M, O'Neill M, Ekárt A, Vanneschi L, Esparcia-Alcázar AI (eds) Proceedings of the 10th European Conference on Genetic Programming, Springer, Valencia, Spain, Lecture Notes in Computer Science, vol 4445, pp 90–101, DOI doi:10.1007/978-3-540-71605-1_9
131. Harrigan GG, LaPlante RH, Cosma GN, Cockerell G, Goodacre R, Maddox JF, Luyendyk JP, Ganey PE, Roth RA (2004) Application of high-throughput fourier-transform infrared spectroscopy in toxicology studies: contribution to a study on the development of an animal model for idiosyncratic toxicity. Toxicology Letters 146(3):197–205, DOI doi:10.1016/j.toxlet.2003.09.011
132. Harris C, Buxton B (1996) GP-COM: A distributed, component-based genetic programming system in C++. In: Koza JR, Goldberg DE, Fogel DB, Riolo RL (eds) Genetic Programming 1996: Proceedings of the First Annual Conference, MIT Press, Stanford University, CA, USA, p 425, URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/gp96com.ps.gz
133. Harvey B, Foster J, Frincke D (1999) Towards byte code genetic programming. In: Banzhaf W, Daida J, Eiben AE, Garzon MH, Honavar V, Jakiela M, Smith RE (eds) Proceedings of the Genetic and Evolutionary Computation Conference, Morgan Kaufmann, Orlando, Florida, USA, vol 2, p 1234, URL http://citeseer.ist.psu.edu/468509.html
134. Hasan S, Daugelat S, Rao PSS, Schreiber M (2006) Prioritizing genomic drug targets in pathogens: Application to mycobacterium tuberculosis. PLoS Computational Biology 2(6):e61, DOI doi:10.1371/journal.pcbi.0020061

135. Hauptman A, Sipper M (2005) GP-endchess: Using genetic programming to evolve chess endgame players. In: Keijzer M, Tettamanzi A, Collet P, van Hemert JI, Tomassini M (eds) Proceedings of the 8th European Conference on Genetic Programming, Springer, Lausanne, Switzerland, Lecture Notes in Computer Science, vol 3447, pp 120–131, URL http://www.cs.bgu.ac.il/~sipper/papabs/eurogpchess-final.pdf
136. Hauptman A, Sipper M (2007) Evolution of an efficient search algorithm for the mate-in-N problem in chess. In: Ebner M, O'Neill M, Ekárt A, Vanneschi L, Esparcia-Alcázar AI (eds) Proceedings of the 10th European Conference on Genetic Programming, Springer, Valencia, Spain, Lecture Notes in Computer Science, vol 4445, pp 78–89, DOI doi:10.1007/978-3-540-71605-1_8
137. Haynes T, Wainwright R, Sen S, Schoenefeld D (1995) Strongly typed genetic programming in evolving cooperation strategies. In: Eshelman L (ed) Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95), Morgan Kaufmann, Pittsburgh, PA, USA, pp 271–278, URL http://www.mcs.utulsa.edu/~rogerw/papers/Haynes-icga95.pdf
138. Haynes TD, Schoenefeld DA, Wainwright RL (1996) Type inheritance in strongly typed genetic programming. In: Angeline PJ, Kinnear, Jr KE (eds) Advances in Genetic Programming 2, MIT Press, Cambridge, MA, USA, chap 18, pp 359–376, URL http://www.mcs.utulsa.edu/~rogerw/papers/Haynes-hier.pdf
139. Heidema AG, Boer JMA, Nagelkerke N, Mariman ECM, van der A DL, Feskens EJM (2006) The challenge for genetic epidemiologists: how to analyze large numbers of SNPs in relation to complex diseases. BMC Genetics 7(23), DOI doi:10.1186/1471-2156-7-23, URL http://www.biomedcentral.com/content/pdf/1471-2156-7-23.pdf
140. Hillis WD (1992) Co-evolving parasites improve simulated evolution as an optimization procedure. In: Langton CG, Taylor CE, Farmer JD, Rasmussen S (eds) Artificial Life II, Santa Fe Institute Studies in the Sciences of Complexity, vol X, Addison-Wesley, Santa Fe Institute, New Mexico, USA, pp 313–324
141. Hinchliffe MP, Willis MJ (2003) Dynamic systems modeling using genetic programming. Computers & Chemical Engineering 27(12):1841–1854, URL http://www.sciencedirect.com/science/article/B6TFT-49MDYGW-2/2/742bcc7f22240c7a0381027aa5ff7e73
142. Ho SY, Hsieh CH, Chen HM, Huang HL (2006) Interpretable gene expression classifier with an accurate and compact fuzzy rule base for microarray data analysis. Biosystems 85(3):165–176, DOI doi:10.1016/j.biosystems.2006.01.002
143. Hoai NX, McKay RI, Abbass HA (2003) Tree adjoining grammars, language bias, and genetic programming. In: Ryan C, Soule T, Keijzer M, Tsang E, Poli R, Costa E (eds) Genetic Programming, Proceedings of EuroGP'2003, Springer-Verlag, Essex, LNCS, vol 2610, pp 335–344, URL http://www.cs.adfa.edu.au/~abbass/publications/hardcopies/TAG3P-EuroGp-03.pdf
144. Hoai NX, McKay RIB, Essam D (2006) Representation and structural difficulty in genetic programming. IEEE Transactions on Evolutionary Computation 10(2):157–166, DOI doi:10.1109/TEVC.2006.871252, URL http://sc.snu.ac.kr/courses/2006/fall/pg/aai/GP/nguyen/Structdiff.pdf
145. Holland J (1975) Adaptation in Natural and Artificial Systems. University of Michigan Press, Ann Arbor, USA

146. Holland JH (1992) Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence. MIT Press, first Published by University of Michigan Press 1975
147. Hong JH, Cho SB (2006) The classification of cancer based on DNA microarray data that uses diverse ensemble genetic programming. Artificial Intelligence In Medicine 36(1):43–58, DOI doi:10.1016/j.artmed.2005.06.002
148. Howard D, Roberts SC (2004) Incident detection on highways. In: O'Reilly UM, Yu T, Riolo RL, Worzel B (eds) Genetic Programming Theory and Practice II, Springer, Ann Arbor, chap 16, pp 263–282
149. Howard D, Roberts SC, Brankin R (1999) Target detection in imagery by genetic programming. Advances in Engineering Software 30(5):303–311, URL http://www.sciencedirect.com/science/article/B6V1P-3W1XV4H-1/1/6e7aee809f33757d0326c62a21824411
150. Howard D, Roberts SC, Ryan C (2006) Pragmatic genetic programming strategy for the problem of vehicle detection in airborne reconnaissance. Pattern Recognition Letters 27(11):1275–1288, DOI doi:10.1016/j.patrec.2005.07.025, evolutionary Computer Vision and Image Understanding
151. Iba H (1996) Genetic Programming. Tokyo Denki University Press
152. Iba H, de Garis H, Sato T (1994) Genetic programming using a minimum description length principle. In: Kinnear, Jr KE (ed) Advances in Genetic Programming, MIT Press, chap 12, pp 265–284, URL http://citeseer.ist.psu.edu/327857.html
153. Inagaki Y (2002) On synchronized evolution of the network of automata. IEEE Transactions on Evolutionary Computation 6(2):147–158, URL http://ieeexplore.ieee.org/iel5/4235/21497/00996014.pdf?tp=&arnumber=996014&isnumber=21497&arSt=147&ared=158&arAuthor=Inagaki%2C+Y.%3B
154. Jacob C (1997) Principia Evolvica – Simulierte Evolution mit Mathematica. dpunkt.verlag, Heidelberg, Germany
155. Jacob C (2000) The art of genetic programming. IEEE Intelligent Systems 15(3):83–84, URL http://ieeexplore.ieee.org/iel5/5254/18363/00846288.pdf
156. Jacob C (2001) Illustrating Evolutionary Computation with Mathematica. Morgan Kaufmann, URL http://www.mkp.com/books_catalog/catalog.asp?ISBN=1-55860-637-8
157. Jeong KS, Kim DK, Whigham P, Joo GJ (2003) Modeling microcystis aeruginosa bloom dynamics in the nakdong river by means of evolutionary computation and statistical approach. Ecological Modeling 161(1–2):67–78, DOI doi:10.1016/S0304-3800(02)00280-6, URL http://www.business.otago.ac.nz/infosci/SIRC/PeterW/Publications/Jeong_EcolMod_V161_Is_1_2_pg67_78.pdf
158. Jin N, Tsang E (2006) Co-adaptive strategies for sequential bargaining problems with discount factors and outside options. In: Proceedings of the 2006 IEEE Congress on Evolutionary Computation, IEEE Press, Vancouver, pp 7913–7920
159. Johnson HE, Gilbert RJ, Winson MK, Goodacre R, Smith AR, Rowland JJ, Hall MA, Kell DB (2000) Explanatory analysis of the metabolome using genetic programming of simple, interpretable rules. Genetic Programming and Evolvable Machines 1(3):243–258, DOI doi:10.1023/A:1010014314078
160. Jones A, Young D, Taylor J, Kell DB, Rowland JJ (1998) Quantification of microbial productivity via multi-angle light scattering and supervised learning. Biotechnology and Bioengineering 59(2):131–143

161. Jordaan E, Kordon A, Chiang L, Smits G (2004) Robust inferential sensors based on ensemble of predictors generated by genetic programming. In: Yao X, Burke E, Lozano JA, Smith J, Merelo-Guervós JJ, Bullinaria JA, Rowe J, Kabán PTA, Schwefel HP (eds) Parallel Problem Solving from Nature - PPSN VIII, Springer-Verlag, Birmingham, UK, LNCS, vol 3242, pp 522–531, DOI doi:10.1007/b100601, URL http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=3242&spage=522
162. Juille H, Pollack JB (1996) Massively parallel genetic programming. In: Angeline PJ, Kinnear, Jr KE (eds) Advances in Genetic Programming 2, MIT Press, Cambridge, MA, USA, chap 17, pp 339–358, URL http://www.demo.cs.brandeis.edu/papers/gp2.pdf
163. Kaboudan M (1999) A measure of time series predictability using genetic programming applied to stock returns. Journal of Forecasting 18:345–357
164. Kaboudan M (2005) Extended daily exchange rates forecasts using wavelet temporal resolutions. New Mathematics and Natural Computing 1:79–107
165. Kaboudan MA (2000) Genetic programming prediction of stock prices. Computational Economics 6(3):207–236
166. Keijzer M (1996) Efficiently representing populations in genetic programming. In: Angeline PJ, Kinnear, Jr KE (eds) Advances in Genetic Programming 2, MIT Press, Cambridge, MA, USA, chap 13, pp 259–278
167. Keijzer M (2004) Scaled symbolic regression. Genetic Programming and Evolvable Machines 5(3):259–269, DOI doi:10.1023/B:GENP.0000030195.77571.f9
168. Kell D (2002a) Defence against the flood. Bioinformatics World pp 16–18, URL http://dbkgroup.org/Papers/biwpp16-18_as_publ.pdf
169. Kell DB (2002b) Genotype-phenotype mapping: genes as computer programs. Trends in Genetics 18(11):555–559, DOI doi:10.1016/S0168-9525(02)02765-8, URL http://dbkgroup.org/Papers/trends_genet_18_(555).pdf
170. Kell DB (2002c) Metabolomics and machine learning: Explanatory analysis of complex metabolome data using genetic programming to produce simple, robust rules. Molecular Biology Reports 29(1–2):237–241, DOI doi:10.1023/A:1020342216314, URL http://dbkgroup.org/Papers/btk2002_dbk.pdf
171. Kell DB, Darby RM, Draper J (2001) Genomic computing. explanatory analysis of plant expression profiling data using machine learning. Plant Physiology 126(3):943–951
172. Keller RE, Poli R (2007a) Cost-benefit investigation of a genetic-programming hyperheuristic. In: Proceedings of Evolution Artificielle
173. Keller RE, Poli R (2007b) Linear genetic programming of metaheuristics. In: Thierens D, Beyer HG, Bongard J, Branke J, Clark JA, Cliff D, Congdon CB, Deb K, Doerr B, Kovacs T, Kumar S, Miller JF, Moore J, Neumann F, Pelikan M, Poli R, Sastry K, Stanley KO, Stutzle T, Watson RA, Wegener I (eds) GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation, ACM Press, London, vol 2, pp 1753–1753, URL http://www.cs.bham.ac.uk/~wbl/biblio/gecco2007/docs/p1753.pdf
174. Keller RE, Poli R (2007c) Linear genetic programming of parsimonious metaheuristics. In: Proceedings of IEEE Congress on Evolutionary Computation (CEC)
175. KHosraviani B (2003) Organization design optimization using genetic programming. In: Koza JR (ed) Genetic Algorithms and Genetic Programming at Stanford 2003, Stanford Bookstore, Stanford, California, 94305-3079 USA, pp 109–117, URL http://www.genetic-programming.org/sp2003/KHosraviani.pdf

176. KHosraviani B, Levitt RE, Koza JR (2004) Organization design optimization using genetic programming. In: Keijzer M (ed) Late Breaking Papers at the 2004 Genetic and Evolutionary Computation Conference, Seattle, Washington, USA, URL http://www.cs.bham.ac.uk/~wbl/biblio/gecco2004/LBP056.pdf
177. Kibria RH, Li Y (2006) Optimizing the initialization of dynamic decision heuristics in DPLL SAT solvers using genetic programming. In: Collet P, Tomassini M, Ebner M, Gustafson S, Ekárt A (eds) Proceedings of the 9th European Conference on Genetic Programming, Springer, Budapest, Hungary, Lecture Notes in Computer Science, vol 3905, pp 331–340, URL http://link.springer.de/link/service/series/0558/papers/3905/39050331.pdf
178. Kinnear, Jr KE (1993) Evolving a sort: Lessons in genetic programming. In: Proceedings of the 1993 International Conference on Neural Networks, IEEE Press, San Francisco, USA, vol 2, pp 881–888, DOI doi:10.1109/ICNN.1993.298674, URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/ftp.io.com/papers/kinnear.icnn93.ps.Z
179. Kinnear, Jr KE (ed) (1994a) Advances in Genetic Programming. MIT Press, Cambridge, MA, URL http://mitpress.mit.edu/book-home.tcl?isbn=0262111888
180. Kinnear, Jr KE (1994b) Fitness landscapes and difficulty in genetic programming. In: Proceedings of the 1994 IEEE World Conference on Computational Intelligence, IEEE Press, Orlando, Florida, USA, vol 1, pp 142–147, DOI doi:10.1109/ICEC.1994.350026, URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/ftp.io.com/papers/kinnear.wcci.ps.Z
181. Kinnear, Jr KE (1994c) A perspective on the work in this book. In: Kinnear, Jr KE (ed) Advances in Genetic Programming, MIT Press, chap 1, pp 3–19, URL http://cognet.mit.edu/library/books/view?isbn=0262111888
182. Klassen TJ, Heywood MI (2002) Towards the on-line recognition of arabic characters. In: Proceedings of the 2002 International Joint Conference on Neural Networks IJCNN'02, IEEE Press, Hilton Hawaiian Village Hotel, Honolulu, Hawaii, pp 1900–1905, URL http://users.cs.dal.ca/~mheywood/X-files/Publications/IEEEarabic.pdf
183. Klein J, Spector L (2007) Unwitting distributed genetic programming via asynchronous javascript and XML. In: Thierens D, Beyer HG, Bongard J, Branke J, Clark JA, Cliff D, Congdon CB, Deb K, Doerr B, Kovacs T, Kumar S, Miller JF, Moore J, Neumann F, Pelikan M, Poli R, Sastry K, Stanley KO, Stutzle T, Watson RA, Wegener I (eds) GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation, ACM Press, London, vol 2, pp 1628–1635, URL http://www.cs.bham.ac.uk/~wbl/biblio/gecco2007/docs/p1628.pdf
184. Kordon A (2006) Evolutionary computation at dow chemical. SIGEVOlution 1(3):4–9, URL http://www.sigevolution.org/2006/03/issue.pdf
185. Kordon A, Castillo F, Smits G, Kotanchek M (2005) Application issues of genetic programming in industry. In: Yu T, Riolo RL, Worzel B (eds) Genetic Programming Theory and Practice III, Genetic Programming, vol 9, Springer, Ann Arbor, chap 16, pp 241–258
186. Kovacic M, Balic J (2003) Evolutionary programming of a CNC cutting machine. International journal for advanced manufacturing technology 22(1–2):118–124, DOI doi:10.1007/s00170-002-1450-8, URL http://www.springerlink.com/openurl.asp?genre=article&eissn=1433-3015&volume=22&issue=1&spage=118

187. Koza JR (1990) A genetic approach to econometric modeling. In: Sixth World Congress of the Econometric Society, Barcelona, Spain, URL http://www.genetic-programming.com/jkpdf/wces1990.pdf
188. Koza JR (1992) Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge, MA, USA
189. Koza JR (1994a) Genetic Programming II: Automatic Discovery of Reusable Programs. MIT Press, Cambridge Massachusetts
190. Koza JR (1994b) Genetic Programming II Videotape: The next generation. MIT Press, 55 Hayward Street, Cambridge, MA, USA
191. Koza JR, Andre D (1996) Classifying protein segments as transmembrane domains using architecture-altering operations in genetic programming. In: Angeline PJ, Kinnear, Jr KE (eds) Advances in Genetic Programming 2, MIT Press, Cambridge, MA, USA, chap 8, pp 155–176, URL http://www.genetic-programming.com/jkpdf/aigp2aatmjk1996.pdf
192. Koza JR, Poli R (2005) Genetic programming. In: Burke EK, Kendall G (eds) Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques, Springer, chap 5, URL http://www.springer.com/sgw/cda/frontpage/0,11855,4-10045-22-67933962-0,00.html
193. Koza JR, Andre D, Bennett III FH, Keane MA (1996a) Use of automatically defined functions and architecture-altering operations in automated circuit synthesis using genetic programming. In: Koza JR, Goldberg DE, Fogel DB, Riolo RL (eds) Genetic Programming 1996: Proceedings of the First Annual Conference, MIT Press, Stanford University, CA, USA, pp 132–149, URL http://www.genetic-programming.com/jkpdf/gp1996adfaa.pdf
194. Koza JR, Bennett III FH, Andre D, Keane MA (1996b) Automated WYWIWYG design of both the topology and component values of electrical circuits using genetic programming. In: Koza JR, Goldberg DE, Fogel DB, Riolo RL (eds) Genetic Programming 1996: Proceedings of the First Annual Conference, MIT Press, Stanford University, CA, USA, pp 123–131, URL http://www.genetic-programming.com/jkpdf/gp1996nielsen.pdf
195. Koza JR, Andre D, Bennett III FH, Keane M (1999a) Genetic Programming 3: Darwinian Invention and Problem Solving. Morgan Kaufman, URL http://www.genetic-programming.org/gpbook3toc.html
196. Koza JR, Bennett III FH, Stiffelman O (1999b) Genetic programming as a Darwinian invention machine. In: Poli R, Nordin P, Langdon WB, Fogarty TC (eds) Genetic Programming, Proceedings of EuroGP'99, Springer-Verlag, Goteborg, Sweden, LNCS, vol 1598, pp 93–108, URL http://www.genetic-programming.com/jkpdf/eurogp1999.pdf
197. Koza JR, Keane MA, Streeter MJ, Mydlowec W, Yu J, Lanza G (2003) Genetic Programming IV: Routine Human-Competitive Machine Intelligence. Kluwer Academic Publishers, URL http://www.genetic-programming.org/gpbook4toc.html
198. Koza JR, Al-Sakran SH, Jones LW (2005) Automated re-invention of six patented optical lens systems using genetic programming. In: Beyer HG, O'Reilly UM, Arnold DV, Banzhaf W, Blum C, Bonabeau EW, Cantu-Paz E, Dasgupta D, Deb K, Foster JA, de Jong ED, Lipson H, Llora X, Mancoridis S, Pelikan M, Raidl GR, Soule T, Tyrrell AM, Watson JP, Zitzler E (eds) GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation, ACM Press, Washington DC, USA, vol 2, pp 1953–1960, URL http://www.cs.bham.ac.uk/~wbl/biblio/gecco2005/docs/p1953.pdf

199. Krasnogor N (2004) Self generating metaheuristics in bioinformatics: The proteins structure comparison case. Genetic Programming and Evolvable Machines 5(2):181–201, DOI doi:10.1023/B:GENP.0000023687.41210.d7
200. Krawiec K (2004) Evolutionary Feature Programming: Cooperative learning for knowledge discovery and computer vision. 385, Wydawnictwo Politechniki Poznanskiej, Poznan University of Technology, Poznan, Poland, URL http://idss.cs.put.poznan.pl/~krawiec/pubs/hab/krawiec_hab.pdf
201. Langdon WB (1996) A bibliography for genetic programming. In: Angeline PJ, Kinnear, Jr KE (eds) Advances in Genetic Programming 2, MIT Press, Cambridge, MA, USA, chap B, pp 507–532, URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/WBL.aigp2.appx.ps.gz
202. Langdon WB (1998a) The evolution of size in variable length representations. In: 1998 IEEE International Conference on Evolutionary Computation, IEEE Press, Anchorage, Alaska, USA, pp 633–638, DOI doi:10.1109/ICEC.1998.700102, URL http://www.cs.bham.ac.uk/~wbl/ftp/papers/WBL.wcci98_bloat.pdf
203. Langdon WB (1998b) Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming!, Genetic Programming, vol 1. Kluwer, Boston, URL http://www.cs.ucl.ac.uk/staff/W.Langdon/gpdata
204. Langdon WB (1999a) Scaling of program tree fitness spaces. Evolutionary Computation 7(4):399–428, URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/WBL.fitnessspaces.pdf
205. Langdon WB (1999b) Size fair and homologous tree genetic programming crossovers. In: Banzhaf W, Daida J, Eiben AE, Garzon MH, Honavar V, Jakiela M, Smith RE (eds) Proceedings of the Genetic and Evolutionary Computation Conference, Morgan Kaufmann, Orlando, Florida, USA, vol 2, pp 1092–1097, URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/WBL.gecco99.fairxo.ps.gz
206. Langdon WB (2000) Size fair and homologous tree genetic programming crossovers. Genetic Programming and Evolvable Machines 1(1/2):95–119, DOI doi:10.1023/A:1010024515191, URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/WBL_fairxo.pdf
207. Langdon WB (2002a) Convergence rates for the distribution of program outputs. In: Langdon WB, Cantú-Paz E, Mathias K, Roy R, Davis D, Poli R, Balakrishnan K, Honavar V, Rudolph G, Wegener J, Bull L, Potter MA, Schultz AC, Miller JF, Burke E, Jonoska N (eds) GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, Morgan Kaufmann Publishers, New York, pp 812–819, URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/wbl_gecco2002.pdf
208. Langdon WB (2002b) How many good programs are there? How long are they? In: De Jong KA, Poli R, Rowe JE (eds) Foundations of Genetic Algorithms VII, Morgan Kaufmann, Torremolinos, Spain, pp 183–202, URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/wbl_foga2002.pdf, published 2003
209. Langdon WB (2003a) Convergence of program fitness landscapes. In: Cantú-Paz E, Foster JA, Deb K, Davis D, Roy R, O'Reilly UM, Beyer HG, Standish R, Kendall G, Wilson S, Harman M, Wegener J, Dasgupta D, Potter MA, Schultz AC, Dowsland K, Jonoska N, Miller J (eds) Genetic and Evolutionary Computation – GECCO-2003, Springer-Verlag, Chicago, LNCS,

vol 2724, pp 1702–1714, URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/wbl_gecco2003.pdf
210. Langdon WB (2003b) The distribution of reversible functions is Normal. In: Riolo RL, Worzel B (eds) Genetic Programming Theory and Practise, Kluwer, chap 11, pp 173–188, URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/wbl_reversible.pdf
211. Langdon WB (2004) Global distributed evolution of L-systems fractals. In: Keijzer M, O'Reilly UM, Lucas SM, Costa E, Soule T (eds) Genetic Programming, Proceedings of EuroGP'2004, Springer-Verlag, Coimbra, Portugal, LNCS, vol 3003, pp 349–358, URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/egp2004_pfeiffer.pdf
212. Langdon WB (2005a) The distribution of amorphous computer outputs. In: Stepney S, Emmott S (eds) The Grand Challenge in Non-Classical Computation: International Workshop, York, UK, URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/grand_2005.pdf
213. Langdon WB (2005b) Pfeiffer – A distributed open-ended evolutionary system. In: Edmonds B, Gilbert N, Gustafson S, Hales D, Krasnogor N (eds) AISB'05: Proceedings of the Joint Symposium on Socially Inspired Computing (METAS 2005), University of Hertfordshire, Hatfield, UK, pp 7–13, URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/wbl_metas2005.pdf, sSAISB 2005 Convention
214. Langdon WB (2006) Mapping non-conventional extensions of genetic programming. In: Calude CS, Dinneen MJ, Paun G, Rozenberg G, Stepney S (eds) Unconventional Computing 2006, Springer-Verlag, York, LNCS, vol 4135, pp 166–180, DOI doi:10.1007/11839132_14, URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/wbl_uc2002.pdf
215. Langdon WB, Banzhaf W (2005) Repeated sequences in linear genetic programming genomes. Complex Systems 15(4):285–306, URL http://www.cs.ucl.ac.uk/staff/rW.Langdon/ftp/papers/wbl_repeat_linear.pdf
216. Langdon WB, Banzhaf W (2007) A SIMD interpreter for genetic programming on GPU graphics cards. In preparation
217. Langdon WB, Buxton BF (2004) Genetic programming for mining DNA chip data from cancer patients. Genetic Programming and Evolvable Machines 5(3):251–257, DOI doi:10.1023/B:GENP.0000030196.55525.f7, URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/wbl_dnachip.pdf
218. Langdon WB, Harrison AP (2008) GP on SPMD parallel graphics hardware for mega bioinformatics data mining, To appear
219. Langdon WB, Nordin P (2001) Evolving hand-eye coordination for a humanoid robot with machine code genetic programming. In: Miller JF, Tomassini M, Lanzi PL, Ryan C, Tettamanzi AGB, Langdon WB (eds) Genetic Programming, Proceedings of EuroGP'2001, Springer-Verlag, Lake Como, Italy, LNCS, vol 2038, pp 313–324, URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/wbl_handeye.ps.gz
220. Langdon WB, Poli R (2008) Mapping non-conventional extensions of geneticprogramming. Natural Computing 7:21–43. Invited contribution to special issue on Unconventional computing
221. Langdon WB, Poli R (1997) Fitness causes bloat. In: Chawdhry PK, Roy R, Pant RK (eds) Soft Computing in Engineering Design and Manufacturing, Springer-Verlag London, pp 13–22, URL http://www.rcs.bham.ac.uk/~wbl/ftp/papers/WBL.bloat_wsc2.ps.gz

222. Langdon WB, Poli R (2002) Foundations of Genetic Programming. Springer-Verlag, URL http://www.cs.ucl.ac.uk/staff/W.Langdon/FOGP/
223. Langdon WB, Poli R (2006a) The halting probability in von Neumann architectures. In: Collet P, Tomassini M, Ebner M, Gustafson S, Ekárt A (eds) Proceedings of the 9th European Conference on Genetic Programming, Springer, Budapest, Hungary, Lecture Notes in Computer Science, vol 3905, pp 225–237, URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/wbl_egp2006.pdf
224. Langdon WB, Poli R (2006b) On turing complete T7 and MISC F–4 program fitness landscapes. In: Arnold DV, Jansen T, Vose MD, Rowe JE (eds) Theory of Evolutionary Algorithms, Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, Dagstuhl, Germany, no. 06061 in Dagstuhl Seminar Proceedings, URL http://drops.dagstuhl.de/opus/volltexte/2006/595, <http://drops.dagstuhl.de/opus/volltexte/2006/595> [date of citation: 2006-01-01]
225. Langdon WB, Soule T, Poli R, Foster JA (1999) The evolution of size and shape. In: Spector L, Langdon WB, O'Reilly UM, Angeline PJ (eds) Advances in Genetic Programming 3, MIT Press, Cambridge, MA, USA, chap 8, pp 163–190, URL http://www.cs.bham.ac.uk/~wbl/aigp3/ch08.pdf
226. Leung KS, Lee KH, Cheang SM (2002) Genetic parallel programming - evolving linear machine codes on a multiple-ALU processor. In: Yaacob S, Nagarajan R, Chekima A (eds) Proceedings of International Conference on Artificial Intelligence in Engineering and Technology - ICAIET 2002, Universiti Malaysia Sabah, pp 207–213
227. Lew TL, Spencer AB, Scarpa F, Worden K, Rutherford A, Hemez F (2006) Identification of response surface models using genetic programming. Mechanical Systems and Signal Processing 20(8):1819–1831, DOI doi:10.1016/j.ymssp.2005.12.003
228. Lewin DR, Lachman-Shalem S, Grosman B (2006) The role of process system engineering (PSE) in integrated circuit (IC) manufacturing. Control Engineering Practice 15(7):793–802, DOI doi:10.1016/j.conengprac.2006.04.003, special Issue on Award Winning Applications, 2005 IFAC World Congress
229. Li L, Jiang W, Li X, Moser KL, Guo Z, Du L, Wang Q, Topol EJ, Wang Q, Rao S (2005) A robust hybrid between genetic algorithm and support vector machine for extracting an optimal feature gene subset. Genomics 85(1):16–23, DOI doi:10.1016/j.ygeno.2004.09.007
230. Linden R, Bhaya A (2007) Evolving fuzzy rules to model gene expression. Biosystems 88(1-2):76–91, DOI doi:10.1016/j.biosystems.2006.04.006
231. 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 the 2004 Genetic and Evolutionary Computation Conference, Seattle, Washington, USA, URL http://www.cs.bham.ac.uk/~wbl/biblio/gecco2004/LBP063.pdf
232. Liu W, Schmidt B (2006) Mapping of hierarchical parallel genetic algorithms for protein folding onto computational grids. IEICE Transactions on Information and Systems E89-D(2):589–596, DOI doi:10.1093/ietisy/e89-d.2.589
233. Lohn J, Hornby G, Linden D (2004) Evolutionary antenna design for a NASA spacecraft. In: O'Reilly UM, Yu T, Riolo RL, Worzel B (eds) Genetic Programming Theory and Practice II, Springer, Ann Arbor, chap 18, pp 301–315

234. Lohn JD, Hornby GS, Linden DS (2005) Rapid re-evolution of an X-band antenna for NASA's space technology 5 mission. In: Yu T, Riolo RL, Worzel B (eds) Genetic Programming Theory and Practice III, Genetic Programming, vol 9, Springer, Ann Arbor, chap 5, pp 65–78
235. Louchet J (2001) Using an individual evolution strategy for stereovision. Genetic Programming and Evolvable Machines 2(2):101–109, DOI doi:10.1023/A:1011544128842
236. Louchet J, Guyon M, Lesot MJ, Boumaza A (2002) Dynamic flies: a new pattern recognition tool applied to stereo sequence processing. Pattern Recognition Letters 23(1–3):335–345, DOI doi:10.1016/S0167-8655(01)00129-5
237. Loviscach J, Meyer-Spradow J (2003) Genetic programming of vertex shaders. In: Chover M, Hagen H, Tost D (eds) Proceedings of EuroMedia 2003, pp 29–31
238. Luke S (1998) Evolving soccerbots: A retrospective. In: Proceedings of the 12th Annual Conference of the Japanese Society for Artificial Intelligence, URL http://www.cs.gmu.edu/~sean/papers/robocupShort.pdf
239. Luke S (2000) Two fast tree-creation algorithms for genetic programming. IEEE Transactions on Evolutionary Computation 4(3):274–283, URL http://ieeexplore.ieee.org/iel5/4235/18897/00873237.pdf
240. Lukschandl E, Borgvall H, Nohle L, Nordahl M, Nordin P (2000) Distributed java bytecode genetic programming. In: Poli R, Banzhaf W, Langdon WB, Miller JF, Nordin P, Fogarty TC (eds) Genetic Programming, Proceedings of EuroGP'2000, Springer-Verlag, Edinburgh, LNCS, vol 1802, pp 316–325, URL http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=1802&spage=316
241. Machado P, Romero J (eds) (2008) The Art of Artificial Evolution. Springer
242. Marenbach P (1998) Using prior knowledge and obtaining process insight in data based modeling of bioprocesses. System Analysis Modeling Simulation 31:39–59
243. Markose S, Tsang E, Er H, Salhi A (2001) Evolutionary arbitrage for FTSE-100 index options and futures. In: Proceedings of the 2001 Congress on Evolutionary Computation CEC2001, IEEE Press, COEX, World Trade Center, 159 Samseong-dong, Gangnam-gu, Seoul, Korea, pp 275–282, DOI doi:10.1109/CEC.2001.934401, URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/TsangCEE2001.pdf
244. Marney JP, Miller D, Fyfe C, Tarbert HFE (2001) Risk adjusted returns to technical trading rules: a genetic programming approach. In: 7th International Conference of Society of Computational Economics, Yale
245. Martin MC (2006) Evolving visual sonar: Depth from monocular images. Pattern Recognition Letters 27(11):1174–1180, DOI doi:10.1016/j.patrec.2005.07.015, URL http://martincmartin.com/papers/EvolvingVisualSonar-PatternRecognitionLetters2006.pdf, evolutionary Computer Vision and Image Understanding
246. Martin P (2001) A hardware implementation of a genetic programming system using FPGAs and Handel-C. Genetic Programming and Evolvable Machines 2(4):317–343, DOI doi:10.1023/A:1012942304464, URL http://www.naiadhome.com/gpem-d.pdf
247. Massey P, Clark JA, Stepney S (2005) Evolution of a human-competitive quantum fourier transform algorithm using genetic programming. In: Beyer HG, O'Reilly UM, Arnold DV, Banzhaf W, Blum C, Bonabeau EW, Cantu-Paz E,

Dasgupta D, Deb K, Foster JA, de Jong ED, Lipson H, Llora X, Mancoridis S, Pelikan M, Raidl GR, Soule T, Tyrrell AM, Watson JP, Zitzler E (eds) GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation, ACM Press, Washington DC, USA, vol 2, pp 1657–1663, URL http://www.cs.bham.ac.uk/~wbl/biblio/gecco2005/docs/p1657.pdf

248. Maxwell III SR (1994) Experiments with a coroutine model for genetic programming. In: Proceedings of the 1994 IEEE World Congress on Computational Intelligence, IEEE Press, Orlando, Florida, USA, vol 1, pp 413–417a, URL http://ieeexplore.ieee.org/iel2/1125/8059/00349915.pdf?isNumber=8059
249. McCormack J (2006) New challenges for evolutionary music and art. SIGEvolution 1(1):5–11, URL http://www.sigevolution.org/2006/01/issue.pdf
250. McGovern AC, Broadhurst D, Taylor J, Kaderbhai N, Winson MK, Small DA, Rowland JJ, Kell DB, Goodacre R (2002) Monitoring of complex industrial bioprocesses for metabolite concentrations using modern spectroscopies and machine learning: Application to gibberellic acid production. Biotechnology and Bioengineering 78(5):527–538, DOI doi:10.1002/bit.10226, URL http://dbkgroup.org/Papers/biotechnol_bioeng_78_(527).pdf
251. McKay B, Willis M, Searson D, Montague G (2000) Nonlinear continuum regression: an evolutionary approach. Transactions of the Institute of Measurement and Control 22(2):125–140, doi:10.1177/014233120002200202, URL http://www.ingentaconnect.com/content/arn/tm/2000/00000022/00000002/art00007
252. McPhee NF, Miller JD (1995) Accurate replication in genetic programming. In: Eshelman L (ed) Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95), Morgan Kaufmann, Pittsburgh, PA, USA, pp 303–309, URL http://www.mrs.umn.edu/~mcphee/Research/Accurate_replication.ps
253. McPhee NF, Hopper NJ, Reierson ML (1998) Sutherland: An extensible object-oriented software framework for evolutionary computation. In: Koza JR, Banzhaf W, Chellapilla K, Deb K, Dorigo M, Fogel DB, Garzon MH, Goldberg DE, Iba H, Riolo R (eds) Genetic Programming 1998: Proceedings of the Third Annual Conference, Morgan Kaufmann, University of Wisconsin, Madison, Wisconsin, USA, p 241, URL http://www.mrs.umn.edu/~mcphee/Research/Sutherland/rsutherland_gp98_announcement.ps.gz
254. Mercure PK, Smits GF, Kordon A (2001) Empirical emulators for first principle models. In: AIChE Fall Annual Meeting, Reno Hilton, URL http://www.aiche.org/conferences/techprogram/paperdetail.asp?PaperID=2373&DSN=annual01
255. Meyer-Spradow J, Loviscach J (2003) Evolutionary design of BRDFs. In: Chover M, Hagen H, Tost D (eds) Eurographics 2003 Short Paper Proceedings, pp 301–306, URL http://viscg.uni-muenster.de/publications/2003/ML03/evolutionary_web.pdf
256. Miller JF (1999) An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach. In: Banzhaf W, Daida J, Eiben AE, Garzon MH, Honavar V, Jakiela M, Smith RE (eds) Proceedings of the Genetic and Evolutionary Computation Conference, Morgan Kaufmann, Orlando, Florida, USA, vol 2, pp 1135–1142, URL http://citeseer.ist.psu.edu/153431.html

257. Miller JF, Smith SL (2006) Redundancy and computational efficiency in cartesian genetic programming. IEEE Transactions on Evolutionary Computation 10(2):167–174, DOI doi:10.1109/TEVC.2006.871253
258. Mitavskiy B, Rowe J (2006) Some results about the markov chains associated to GPs and to general EAs. Theoretical Computer Science 361(1):72–110, DOI doi:10.1016/j.tcs.2006.04.006
259. Montana DJ (1995) Strongly typed genetic programming. Evolutionary Computation 3(2):199–230, URL http://vishnu.bbn.com/papers/stgp.pdf
260. Moore GE (1965) Cramming more components onto integrated circuits. Electronics 38(8):114–117
261. Moore JH, Parker JS, Olsen NJ, Aune TM (2002) Symbolic discriminant analysis of microarray data in automimmune disease. Genetic Epidemiology 23:57–69
262. Motsinger AA, Lee SL, Mellick G, Ritchie MD (2006) GPNN: Power studies and applications of a neural network method for detecting gene-gene interactions in studies of human disease. BMC bioinformatics [electronic resource] 7(1):39–39, DOI doi:10.1186/1471-2105-7-39, URL http://www.biomedcentral.com/1471-2105/7/39
263. Neely CJ (2003) Risk-adjusted, ex ante, optimal technical trading rules in equity markets. International Review of Economics and Finance 12(1):69–87, DOI doi:10.1016/S1059-0560(02)00129-6, URL http://research.stlouisfed.org/wp/1999/1999-015.pdf
264. Neely CJ, Weller PA (1999) Technical trading rules in the european monetary system. Journal of International Money and Finance 18(3):429–458, DOI doi:10.1016/S0261-5606(99)85005-0, URL http://research.stlouisfed.org/wp/1997/97-015.pdf
265. Neely CJ, Weller PA (2001a) Predicting exchange rate volatility: Genetic programming vs. GARCH and risk metrics. Working Paper 2001-009B, Economic, Research, Federal Reserve Bank of St. Louis, 411 Locust Street, St. Louis, MO 63102-0442, USA, URL http://research.stlouisfed.org/wp/2001/2001-009.pdf
266. Neely CJ, Weller PA (2001b) Technical analysis and central bank intervention. Journal of International Money and Finance 20(7):949–970, DOI doi:10.1016/S0261-5606(01)00033-X, URL http://research.stlouisfed.org/wp/1997/97-002.pdf
267. Neely CJ, Weller PA, Dittmar R (1997) Is technical analysis in the foreign exchange market profitable? A genetic programming approach. The Journal of Financial and Quantitative Analysis 32(4):405–426, URL http://links.jstor.org/sici?sici=0022-1090%28199712%2932%3A4%3C405%3AITAITF%3E2.0.CO%3B2-T
268. Neely CJ, Weller PA, Ulrich JM (2006) The adaptive markets hypothesis: evidence from the foreign exchange market. Working Paper 2006-046B, Federal Reserve Bank of St. Louis, Research Division, P.O. Box 442, St. Louis, MO 63166, USA, URL http://research.stlouisfed.org/wp/2006/2006-046.pdf, revised March 2007
269. Nikolaev N, Iba H (2006) Adaptive Learning of Polynomial Networks Genetic Programming, Backpropagation and Bayesian Methods. No. 4 in Genetic and Evolutionary Computation, Springer, june
270. Nikolaev NY, Iba H (2002) Genetic programming of polynomial models for financial forecasting. In: Chen SH (ed) Genetic Algorithms and Genetic

Programming in Computational Finance, Kluwer Academic Press, chap 5, pp 103–123
271. Nix AE, Vose MD (1992) Modeling genetic algorithms with Markov chains. Annals of Mathematics and Artificial Intelligence 5:79–88
272. Nordin P (1994) A compiling genetic programming system that directly manipulates the machine code. In: Kinnear, Jr KE (ed) Advances in Genetic Programming, MIT Press, chap 14, pp 311–331, URL http://cognet.mit.edu/library/books/view?isbn=0262111888
273. Nordin P (1997) Evolutionary program induction of binary machine code and its applications. PhD thesis, der Universitat Dortmund am Fachereich Informatik
274. Nordin P, Johanna W (2003) Humanoider: Sjavlarande robotar och artificiell intelligens. Liber
275. Nordin P, Banzhaf W, Francone FD (1999) Efficient evolution of machine code for CISC architectures using instruction blocks and homologous crossover. In: Spector L, Langdon WB, O'Reilly UM, Angeline PJ (eds) Advances in Genetic Programming 3, MIT Press, Cambridge, MA, USA, chap 12, pp 275–299, URL http://www.aimlearning.com/aigp31.pdf
276. Oakley H (1994) Two scientific applications of genetic programming: Stack filters and non-linear equation fitting to chaotic data. In: Kinnear, Jr KE (ed) Advances in Genetic Programming, MIT Press, chap 17, pp 369–389, URL http://cognet.mit.edu/library/books/view?isbn=0262111888
277. Oltean M (2005) Evolving evolutionary algorithms using linear genetic programming. Evolutionary Computation 13(3):387–410, DOI doi:10.1162/1063656054794815, URL http://www.ingentaconnect.com/content/mitpress/evco/2005/00000013/00000003/art00006
278. Oltean M, Dumitrescu D (2004) Evolving TSP heuristics using multi expression programming. In: Bubak M, van Albada GD, Sloot PMA, Dongarra J (eds) Computational Science - ICCS 2004: 4th International Conference, Part II, Springer-Verlag, Krakow, Poland, Lecture Notes in Computer Science, vol 3037, pp 670–673, DOI doi:10.1007/b97988, URL http://springerlink.metapress.com/openurl.asp?genre=article&issn=0302-9743&volume=3037&spage=670
279. O'Neill M, Ryan C (2003) Grammatical Evolution: Evolutionary Automatic Programming in a Arbitrary Language, Genetic programming, vol 4. Kluwer Academic Publishers, URL http://www.wkap.nl/prod/b/1-4020-7444-1
280. Openshaw S, Turton I (1994) Building new spatial interaction models using genetic programming. In: Fogarty TC (ed) Evolutionary Computing, Springer-Verlag, Leeds, UK, Lecture Notes in Computer Science, URL http://www.geog.leeds.ac.uk/papers/94-1/94-1.pdf
281. O'Reilly UM (1995) An analysis of genetic programming. PhD thesis, Carleton University, Ottawa-Carleton Institute for Computer Science, Ottawa, Ontario, Canada, URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/oreilly/abstract.ps.gz
282. O'Reilly UM, Hemberg M (2007) Integrating generative growth and evolutionary computation for form exploration. Genetic Programming and Evolvable Machines 8(2):163–186, DOI doi:10.1007/s10710-007-9025-y, special issue on developmental systems
283. O'Reilly UM, Oppacher F (1994) The troubling aspects of a building block hypothesis for genetic programming. In: Whitley LD, Vose MD (eds)

Foundations of Genetic Algorithms 3, Morgan Kaufmann, Estes Park, Colorado, USA, pp 73–88, URL http://citeseer.ist.psu.edu/cache/papers/cs/163/http:zSzzSzwww.ai.mit.eduzSzpeoplezSzunamayzSzpaperszSzfoga.pdf/oreilly92troubling.pdf, published 1995

284. O'Reilly UM, Yu T, Riolo RL, Worzel B (eds) (2004) Genetic Programming Theory and Practice II, Genetic Programming, vol 8, Springer, Ann Arbor, MI, USA, URL http://www.springeronline.com/sgw/cda/frontpage/0,11855,5-40356-22-34954683-0,00.html
285. Oussaidène M, Chopard B, Pictet OV, Tomassini M (1997) Parallel genetic programming and its application to trading model induction. Parallel Computing 23(8):1183–1198, URL http://citeseer.ist.psu.edu/cache/papers/cs/166/http:zSzzSzlslwww.epfl.chzSz marcozSzparcomp.pdf/oussaidene97parallel.pdf
286. Owens JD, David, Govindaraju N, Harris M, Kruger J, Lefohn AE, Purcell TJ (2007) A survey of general-purpose computation on graphics hardware. Computer Graphics Forum 26(1):80–113, DOI doi:10.1111/j.1467-8659.2007.01012.x
287. Parrott D, Li X, Ciesielski V (2005) Multi-objective techniques in genetic programming for evolving classifiers. In: Corne D, Michalewicz Z, Dorigo M, Eiben G, Fogel D, Fonseca C, Greenwood G, Chen TK, Raidl G, Zalzala A, Lucas S, Paechter B, Willies J, Guervos JJM, Eberbach E, McKay B, Channon A, Tiwari A, Volkert LG, Ashlock D, Schoenauer M (eds) Proceedings of the 2005 IEEE Congress on Evolutionary Computation, IEEE Press, Edinburgh, UK, vol 2, pp 1141–1148, URL http://goanna.cs.rmit.edu.au/~xiaodong/publications/183.pdf
288. Perkis T (1994) Stack-based genetic programming. In: Proceedings of the 1994 IEEE World Congress on Computational Intelligence, IEEE Press, Orlando, Florida, USA, vol 1, pp 148–153, URL http://citeseer.ist.psu.edu/432690.html
289. Pillay N (2003) Evolving solutions to ASCII graphics programming problems in intelligent programming tutors. In: Akerkar R (ed) International Conference on Applied Artificial Intelligence (ICAAI'2003), TMRF, Fort Panhala, Kolhapur, India, pp 236–243
290. Poli R (1996a) Discovery of symbolic, neuro-symbolic and neural networks with parallel distributed genetic programming. Tech. Rep. CSRP-96-14, University of Birmingham, School of Computer Science, URL ftp://ftp.cs.bham.ac.uk/pub/tech-reports/1996/CSRP-96-14.ps.gz, presented at 3rd International Conference on Artificial Neural Networks and Genetic Algorithms, ICANNGA'97
291. Poli R (1996b) Genetic programming for image analysis. In: Koza JR, Goldberg DE, Fogel DB, Riolo RL (eds) Genetic Programming 1996: Proceedings of the First Annual Conference, MIT Press, Stanford University, CA, USA, pp 363–368, URL http://cswww.essex.ac.uk/staff/rpoli/papers/Poli-GP1996.pdf
292. Poli R (1999) Parallel distributed genetic programming. In: Corne D, Dorigo M, Glover F (eds) New Ideas in Optimization, Advanced Topics in Computer Science, McGraw-Hill, Maidenhead, Berkshire, England, chap 27, pp 403–431, URL http://citeseer.ist.psu.edu/328504.html
293. Poli R (2000a) Exact schema theorem and effective fitness for GP with one-point crossover. In: Whitley D, Goldberg D, Cantu-Paz E, Spector L, Parmee I, Beyer HG (eds) Proceedings of the Genetic and Evolutionary Computation Conference, Morgan Kaufmann, Las Vegas, pp 469–476

294. Poli R (2000b) Hyperschema theory for GP with one-point crossover, building blocks, and some new results in GA theory. In: Poli R, Banzhaf W, Langdon WB, Miller JF, Nordin P, Fogarty TC (eds) Genetic Programming, Proceedings of EuroGP'2000, Springer-Verlag, Edinburgh, LNCS, vol 1802, pp 163–180, URL http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=1802&spage=163
295. Poli R (2001) Exact schema theory for genetic programming and variable-length genetic algorithms with one-point crossover. Genetic Programming and Evolvable Machines 2(2):123–163
296. Poli R (2003) A simple but theoretically-motivated method to control bloat in genetic programming. In: Ryan C, Soule T, Keijzer M, Tsang E, Poli R, Costa E (eds) Genetic Programming, Proceedings of EuroGP'2003, Springer-Verlag, Essex, LNCS, vol 2610, pp 204–217, URL http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=2610&spage=204
297. Poli R (2005) Tournament selection, iterated coupon-collection problem, and backward-chaining evolutionary algorithms. In: Wright AH, Vose MD, De Jong KA, Schmitt LM (eds) Foundations of Genetic Algorithms 8, Springer-Verlag, Aizu-Wakamatsu City, Japan, Lecture Notes in Computer Science, vol 3469, pp 132–155, URL http://www.cs.essex.ac.uk/staff/rpoli/papers/foga2005_Poli.pdf
298. Poli R, Langdon WB (1997) A new schema theory for genetic programming with one-point crossover and point mutation. In: Koza JR, Deb K, Dorigo M, Fogel DB, Garzon M, Iba H, Riolo RL (eds) Genetic Programming 1997: Proceedings of the Second Annual Conference, Morgan Kaufmann, Stanford University, CA, USA, pp 278–285, URL http://citeseer.ist.psu.edu/327495.html
299. Poli R, Langdon WB (1998a) On the search properties of different crossover operators in genetic programming. In: Koza JR, Banzhaf W, Chellapilla K, Deb K, Dorigo M, Fogel DB, Garzon MH, Goldberg DE, Iba H, Riolo R (eds) Genetic Programming 1998: Proceedings of the Third Annual Conference, Morgan Kaufmann, University of Wisconsin, Madison, Wisconsin, USA, pp 293–301, URL http://www.cs.essex.ac.uk/staff/poli/papers/Poli-GP1998.pdf
300. Poli R, Langdon WB (1998b) Schema theory for genetic programming with one-point crossover and point mutation. Evolutionary Computation 6(3):231–252, URL http://cswww.essex.ac.uk/staff/poli/papers/Poli-ECJ1998.pdf
301. Poli R, Langdon WB (2005a) Running genetic programming backward. In: Riolo RL, Worzel B, Yu T (eds) Genetic Programming Theory and Practice, Kluwer
302. Poli R, Langdon WB (2005b) Running genetic programming backward. In: Yu T, Riolo RL, Worzel B (eds) Genetic Programming Theory and Practice III, Genetic Programming, vol 9, Springer, Ann Arbor, chap 9, pp 125–140, URL http://www.cs.essex.ac.uk/staff/poli/papers/GPTP2005.pdf
303. Poli R, Langdon WB (2006a) Backward-chaining evolutionary algorithms. Artificial Intelligence 170(11):953–982, DOI doi:10.1016/j.artint.2006.04.003, URL http://www.cs.essex.ac.uk/staff/poli/papers/aijournal2006.pdf
304. Poli R, Langdon WB (2006b) Efficient markov chain model of machine code program execution and halting. In: Riolo RL, Soule T, Worzel B (eds) Genetic Programming Theory and Practice IV, Genetic and Evolutionary Computation, vol 5, Springer, Ann Arbor, chap 13, URL http://www.cs.essex.ac.uk/staff/poli/papers/GPTP2006.pdf

305. Poli R, McPhee NF (2003a) General schema theory for genetic programming with subtree-swapping crossover: Part I. Evolutionary Computation 11(1):53–66, DOI doi:10.1162/106365603321829005, URL http://cswww.essex.ac.uk/staff/rpoli/papers/ecj2003partI.pdf
306. Poli R, McPhee NF (2003b) General schema theory for genetic programming with subtree-swapping crossover: Part II. Evolutionary Computation 11(2):169–206, DOI doi:10.1162/106365603766646825, URL http://cswww.essex.ac.uk/staff/rpoli/papers/ecj2003partII.pdf
307. Poli R, Page J, Langdon WB (1999) Smooth uniform crossover, sub-machine code GP and demes: A recipe for solving high-order boolean parity problems. In: Banzhaf W, Daida J, Eiben AE, Garzon MH, Honavar V, Jakiela M, Smith RE (eds) Proceedings of the Genetic and Evolutionary Computation Conference, Morgan Kaufmann, Orlando, Florida, USA, vol 2, pp 1162–1169, URL http://www.cs.bham.ac.uk/~wbl/biblio/gecco1999/GP-466.pdf
308. Poli R, Rowe JE, McPhee NF (2001) Markov chain models for GP and variable-length GAs with homologous crossover. In: Spector L, Goodman ED, Wu A, Langdon WB, Voigt HM, Gen M, Sen S, Dorigo M, Pezeshk S, Garzon MH, Burke E (eds) Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001), Morgan Kaufmann, San Francisco, California, USA, pp 112–119, URL http://www.cs.bham.ac.uk/~wbl/biblio/gecco2001/d01.pdf
309. 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. Genetic Programming and Evolvable Machines 5(1):31–70, DOI doi:10.1023/B:GENP.0000017010.41337.a7, URL http://cswww.essex.ac.uk/staff/rpoli/papers/GPEM2004.pdf
310. Poli R, Di Chio C, Langdon WB (2005a) Exploring extended particle swarms: a genetic programming approach. In: Beyer HG, O'Reilly UM, Arnold DV, Banzhaf W, Blum C, Bonabeau EW, Cantu-Paz E, Dasgupta D, Deb K, Foster JA, de Jong ED, Lipson H, Llora X, Mancoridis S, Pelikan M, Raidl GR, Soule T, Tyrrell AM, Watson JP, Zitzler E (eds) GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation, ACM Press, Washington DC, USA, vol 1, pp 169–176, URL http://www.cs.essex.ac.uk/staff/poli/papers/geccopso2005.pdf
311. Poli R, Langdon WB, Holland O (2005b) Extending particle swarm optimisation via genetic programming. In: Keijzer M, Tettamanzi A, Collet P, van Hemert JI, Tomassini M (eds) Proceedings of the 8th European Conference on Genetic Programming, Springer, Lausanne, Switzerland, Lecture Notes in Computer Science, vol 3447, pp 291–300, URL http://www.cs.essex.ac.uk/staff/poli/papers/eurogpPSO2005.pdf
312. Poli R, Langdon WB, Dignum S (2007a) On the limiting distribution of program sizes in tree-based genetic programming. In: Ebner M, O'Neill M, Ekárt A, Vanneschi L, Esparcia-Alcázar AI (eds) Proceedings of the 10th European Conference on Genetic Programming, Springer, Valencia, Spain, Lecture Notes in Computer Science, vol 4445, pp 193–204, DOI doi:10.1007/978-3-540-71605-1_18
313. Poli R, Woodward J, Burke E (2007b) A histogram-matching approach to the evolution of bin-packing strategies. In: Proceedings of the IEEE Congress on Evolutionary Computation, Singapore, accepted

314. Potter MA (1997) The design and analysis of a computational model of cooperative coevolution. PhD thesis, George Mason University, Washington, DC, URL http://www.cs.gmu.edu/~mpotter/dissertation.html
315. Priesterjahn S, Kramer O, Weimer A, Goebels A (2006) Evolution of human-competitive agents in modern computer games. In: Yen GG, Lucas SM, Fogel G, Kendall G, Salomon R, Zhang BT, Coello CAC, Runarsson TP (eds) Proceedings of the 2006 IEEE Congress on Evolutionary Computation, IEEE Press, Vancouver, BC, Canada, pp 777–784, URL http://ieeexplore.ieee.org/servlet/opac?punumber=11108
316. Prügel-Bennett A, Shapiro JL (1994) An analysis of genetic algorithms using statistical mechanics. Physical Review Letters 72:1305–1309
317. Quintana MI, Poli R, Claridge E (2006) Morphological algorithm design for binary images using genetic programming. Genetic Programming and Evolvable Machines 7(1):81–102, DOI doi:10.1007/s10710-006-7012-3, URL http://cswww.essex.ac.uk/staff/rpoli/papers/gpem2005.pdf
318. Ratle A, Sebag M (2000) Genetic programming and domain knowledge: Beyond the limitations of grammar-guided machine discovery. In: Schoenauer M, Deb K, Rudolph G, Yao X, Lutton E, Merelo JJ, Schwefel HP (eds) Parallel Problem Solving from Nature - PPSN VI 6th International Conference, Springer Verlag, Paris, France, LNCS, vol 1917, pp 211–220, URL http://www.lri.fr/~sebag/REF/PPSN00.ps
319. Reggia J, Tagamets M, Contreras-Vidal J, Jacobs D, Weems S, Naqvi W, Winder R, Chabuk T, Jung J, Yang C (2006) Development of a large-scale integrated neurocognitive architecture - part 2: Design and architecture. Tech. Rep. TR-CS-4827, UMIACS-TR-2006-43, University of Maryland, USA, URL https://drum.umd.edu/dspace/bitstream/1903/3957/1/MarylandPart2.pdf
320. Reif DM, White BC, Moore JH (2004) Integrated analysis of genetic, genomic, and proteomic data. Expert Review of Proteomics 1(1):67–75, DOI doi:10.1586/14789450.1.1.67, URL http://www.future-drugs.com/doi/abs/10.1586/14789450.1.1.67
321. Reynolds CW (1987) Flocks, herds, and schools: A distributed behavioral model. SIGGRAPH Computer Graphics 21(4):25–34, URL http://www.red3d.com/cwr/papers/1987/boids.html
322. Riolo RL, Worzel B (2003) Genetic Programming Theory and Practice, Genetic Programming, vol 6. Kluwer, Boston, MA, USA, URL http://www.wkap.nl/prod/b/1-4020-7581-2, series Editor - John Koza
323. Riolo RL, Soule T, Worzel B (eds) (2007a) Genetic Programming Theory and Practice IV, Genetic and Evolutionary Computation, vol 5, Springer, Ann Arbor, URL http://www.springer.com/west/home/computer/foundations?SGWID=%4-156-22-173660377-0
324. Riolo RL, Soule T, Worzel B (eds) (2007b) Genetic Programming Theory and Practice V, Genetic and Evolutionary Computation, Springer, Ann Arbor
325. Ritchie MD, White BC, Parker JS, Hahn LW, Moore JH (2003) Optimization of neural network architecture using genetic programming improves detection and modeling of gene-gene interactions in studies of human diseases. BMC Bioinformatics 4(28), DOI doi:10.1186/1471-2105-4-28, URL http://www.biomedcentral.com/1471-2105/4/28
326. Ritchie MD, Motsinger AA, Bush WS, Coffey CS, Moore JH (2007) Genetic programming neural networks: A powerful bioinformatics tool for human

genetics. Applied Soft Computing 7(1):471–479, DOI doi:10.1016/j.asoc.2006.01.013

327. Rivero D, nal JRR, Dorado J, Pazos A (2004) Using genetic programming for character discrimination in damaged documents. In: Raidl GR, Cagnoni S, Branke J, Corne DW, Drechsler R, Jin Y, Johnson CR, Machado P, Marchiori E, Rothlauf F, Smith GD, Squillero G (eds) Applications of Evolutionary Computing, EvoWorkshops2004: EvoBIO, EvoCOMNET, EvoHOT, EvoIASP, EvoMUSART, EvoSTOC, Springer Verlag, Coimbra, Portugal, LNCS, vol 3005, pp 349–358
328. Robinson A, Spector L (2002) Using genetic programming with multiple data types and automatic modularization to evolve decentralized and coordinated navigation in multi-agent systems. In: Cantú-Paz E (ed) Late Breaking Papers at the Genetic and Evolutionary Computation Conference (GECCO-2002), AAAI, New York, NY, pp 391–396
329. Rodriguez-Vazquez K, Fonseca CM, Fleming PJ (2004) Identifying the structure of nonlinear dynamic systems using multiobjective genetic programming. IEEE Transactions on Systems, Man and Cybernetics, Part A 34(4):531–545
330. Rosca JP (1997) Analysis of complexity drift in genetic programming. In: Koza JR, Deb K, Dorigo M, Fogel DB, Garzon M, Iba H, Riolo RL (eds) Genetic Programming 1997: Proceedings of the Second Annual Conference, Morgan Kaufmann, Stanford University, CA, USA, pp 286–294, URL ftp://ftp.cs.rochester.edu/pub/u/rosca/gp/97.gp.ps.gz
331. Rosca JP, Ballard DH (1996) Discovery of subroutines in genetic programming. In: Angeline PJ, Kinnear, Jr KE (eds) Advances in Genetic Programming 2, MIT Press, Cambridge, MA, USA, chap 9, pp 177–202, URL ftp://ftp.cs.rochester.edu/pub/u/rosca/gp/96.aigp2.dsgp.ps.gz
332. Ross BJ, Gualtieri AG, Fueten F, Budkewitsch P (2005) Hyperspectral image analysis using genetic programming. Applied Soft Computing 5(2):147–156, DOI doi:10.1016/j.asoc.2004.06.003, URL http://www.cosc.brocku.ca/~bross/research/gp_hyper.pdf
333. Rothlauf F (2006) Representations for genetic and evolutionary algorithms, 2nd edn. Springer-Verlag, pub-SV:adr, URL http://download-ebook.org/index.php?target=desc&ebookid=5771, first published 2002, 2nd edition available electronically
334. Ryan C (1999) Automatic Re-engineering of Software Using Genetic Programming, Genetic Programming, vol 2. Kluwer Academic Publishers, URL http://www.wkap.nl/book.htm/0-7923-8653-1
335. Ryan C, Ivan L (1999) An automatice software re-engineering tool based on genetic programming. In: Spector L, Langdon WB, O'Reilly UM, Angeline PJ (eds) Advances in Genetic Programming 3, MIT Press, Cambridge, MA, USA, Ann Arbor, URL http://www.cs.bham.ac.uk/~wbl/aigp3/ch02.pdf
336. Ryan C, Collins JJ, O'Neill M (1998) Grammatical evolution: Evolving programs for an arbitrary language. In: Banzhaf W, Poli R, Schoenauer M, Fogarty TC (eds) Proceedings of the First European Workshop on Genetic Programming, Springer-Verlag, Paris, LNCS, vol 1391, pp 83–95, URL http://www.lania.mx/~ccoello/eurogp98.ps.gz
337. Samuel AL (1983) AI, where it has been and where it is going. In: IJCAI, pp 1152–1157

338. Schmidt MD, Lipson H (2006) Co-evolving fitness predictors for accelerating and reducing evaluations. In: Riolo RL, Soule T, Worzel B (eds) Genetic Programming Theory and Practice IV, Genetic and Evolutionary Computation, vol 5, Springer, Ann Arbor
339. Schoenauer M, Sebag M (2001) Using domain knowledge in evolutionary system identification. In: Giannakoglou KC, Tsahalis D, Periaux J, Papailiou K, Fogarty TC (eds) Evolutionary Methods for Design, Optimization and Control with Applications to Industrial Problems, Athens
340. Schoenauer M, Lamy B, Jouve F (1995) Identification of mechanical behavior by genetic programming part II: Energy formulation. Tech. rep., Ecole Polytechnique, 91128 Palaiseau, France
341. Schoenauer M, Sebag M, Jouve F, Lamy B, Maitournam H (1996) Evolutionary identification of macro-mechanical models. In: Angeline PJ, Kinnear, Jr KE (eds) Advances in Genetic Programming 2, MIT Press, Cambridge, MA, USA, chap 23, pp 467–488, URL http://citeseer.ist.psu.edu/cache/papers/cs/902/http:zSzzSzwww.eeaax.polytechnique.frzSzpaperszSzmarczSzAGP2.pdf/schoenauer96evolutionary.pdf
342. Searson DP, Montague GA, Willis MJ (1998) Evolutionary design of process controllers. In: In Proceedings of the 1998 United Kingdom Automatic Control Council International Conference on Control (UKACC International Conference on Control '98), Institution of Electrical Engineers (IEE), University of Wales, Swansea, UK, IEE Conference Publications, vol 455, URL http://www.staff.ncl.ac.uk/d.p.searson/docs/Searsoncontrol98.pdf
343. Sekanina L (2003) Evolvable Components: From Theory to Hardware Implementations. Natural Computing, Springer-Verlag, URL http://www.fit.vutbr.cz/~sekanina/rehw/books.html.en
344. Setzkorn C (2005) On the use of multi-objective evolutionary algorithms for classification rule induction. PhD thesis, University of Liverpool, UK
345. Shah SC, Kusiak A (2004) Data mining and genetic algorithm based gene/SNP selection. Artificial Intelligence in Medicine 31(3):183–196, DOI doi:10.1016/j.artmed.2004.04.002, URL http://www.icaen.uiowa.edu/~ankusiak/Journal-papers/Gen_Shital.pdf
346. Sharabi S, Sipper M (2006) GP-sumo: Using genetic programming to evolve sumobots. Genetic Programming and Evolvable Machines 7(3):211–230, DOI doi:10.1007/s10710-006-9006-6
347. Sharman KC, Esparcia-Alcazar AI (1993) Genetic evolution of symbolic signal models. In: Proceedings of the Second International Conference on Natural Algorithms in Signal Processing, NASP'93, Essex University, UK, URL http://www.iti.upv.es/~anna/papers/natalg93.ps
348. Sharman KC, Esparcia Alcazar AI, Li Y (1995) Evolving signal processing algorithms by genetic programming. In: Zalzala AMS (ed) First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications, GALESIA, IEE, Sheffield, UK, vol 414, pp 473–480, URL http://www.iti.upv.es/~anna/papers/galesi95.ps
349. Shaw AD, Winson MK, Woodward AM, McGovern AC, Davey HM, Kaderbhai N, Broadhurst D, Gilbert RJ, Taylor J, Timmins EM, Goodacre R, Kell DB, Alsberg BK, Rowland JJ (2000) Bioanalysis and biosensors for bioprocess monitoring rapid analysis of high-dimensional bioprocesses using multivariate spectroscopies and advanced chemometrics. Advances in Biochemical

Engineering/Biotechnology 66:83–113, URL http://www.springerlink.com/link.asp?id=t8b4ya0bl42jnjj3
350. Shichel Y, Ziserman E, Sipper M (2005) GP-robocode: Using genetic programming to evolve robocode players. In: Keijzer M, Tettamanzi A, Collet P, van Hemert JI, Tomassini M (eds) Proceedings of the 8th European Conference on Genetic Programming, Springer, Lausanne, Switzerland, Lecture Notes in Computer Science, vol 3447, pp 143–154, URL http://www.cs.bgu.ac.il/~sipper/papabs/eurogprobo-final.pdf
351. Si HZ, Wang T, Zhang KJ, Hu ZD, Fan BT (2006) QSAR study of 1,4-dihydropyridine calcium channel antagonists based on gene expression programming. Bioorganic & Medicinal Chemistry 14(14):4834–4841, DOI doi:10.1016/j.bmc.2006.03.019
352. Siegel EV (1994) Competitively evolving decision trees against fixed training cases for natural language processing. In: Kinnear, Jr KE (ed) Advances in Genetic Programming, MIT Press, chap 19, pp 409–423, URL http://www1.cs.columbia.edu/nlp/papers/1994/siegel_94.pdf
353. Sims K (1991) Artificial evolution for computer graphics. ACM Computer Graphics 25(4):319–328, URL http://delivery.acm.org/10.1145/130000/122752/p319-sims.pdf, sIGGRAPH '91 Proceedings
354. Smart W, Zhang M (2004) Applying online gradient descent search to genetic programming for object recognition. In: Hogan J, Montague P, Purvis M, Steketee C (eds) CRPIT '04: Proceedings of the second workshop on Australasian information security, Data Mining and Web Intelligence, and Software Internationalisation, Australian Computer Society, Inc., Dunedin, New Zealand, vol 32 no. 7, pp 133–138, URL http://crpit.com/confpapers/CRPITV32Smart.pdf
355. Soule T, Foster JA (1998a) Effects of code growth and parsimony pressure on populations in genetic programming. Evolutionary Computation 6(4):293–309, URL http://mitpress.mit.edu/journals/EVCO/Soule.pdf
356. Soule T, Foster JA (1998b) Removal bias: a new cause of code growth in tree based evolutionary programming. In: 1998 IEEE International Conference on Evolutionary Computation, IEEE Press, Anchorage, Alaska, USA, pp 781–186, URL http://citeseer.ist.psu.edu/313655.html
357. Spector L (2001) Autoconstructive evolution: Push, pushGP, and pushpop. In: Spector L, Goodman ED, Wu A, Langdon WB, Voigt HM, Gen M, Sen S, Dorigo M, Pezeshk S, Garzon MH, Burke E (eds) Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001), Morgan Kaufmann, San Francisco, California, USA, pp 137–146, URL http://hampshire.edu/lspector/pubs/ace.pdf
358. Spector L (2004) Automatic Quantum Computer Programming: A Genetic Programming Approach, Genetic Programming, vol 7. Kluwer Academic Publishers, Boston/Dordrecht/New York/London, URL http://www.wkap.nl/prod/b/1-4020-7894-3
359. Spector L, Alpern A (1994) Criticism, culture, and the automatic generation of artworks. In: Proceedings of Twelfth National Conference on Artificial Intelligence, AAAI Press/MIT Press, Seattle, Washington, USA, pp 3–8
360. Spector L, Alpern A (1995) Induction and recapitulation of deep musical structure. In: Proceedings of International Joint Conference on Artificial Intelligence, IJCAI'95 Workshop on Music and AI, Montreal, Quebec, Canada, URL http://hampshire.edu/lspector/pubs/IJCAI95mus-toappear.ps

361. Spector L, Barnum H, Bernstein HJ (1998) Genetic programming for quantum computers. In: Koza JR, Banzhaf W, Chellapilla K, Deb K, Dorigo M, Fogel DB, Garzon MH, Goldberg DE, Iba H, Riolo R (eds) Genetic Programming 1998: Proceedings of the Third Annual Conference, Morgan Kaufmann, University of Wisconsin, Madison, Wisconsin, USA, pp 365–373
362. Spector L, Barnum H, Bernstein HJ, Swamy N (1999a) Finding a better-than-classical quantum AND/OR algorithm using genetic programming. In: Angeline PJ, Michalewicz Z, Schoenauer M, Yao X, Zalzala A (eds) Proceedings of the Congress on Evolutionary Computation, IEEE Press, Mayflower Hotel, Washington D.C., USA, vol 3, pp 2239–2246, URL http://hampshire.edu/~lasCCS/pubs/spector-cec99.ps
363. Spector L, Langdon WB, O'Reilly UM, Angeline PJ (eds) (1999b) Advances in Genetic Programming 3. MIT Press, Cambridge, MA, USA, URL http://www.cs.bham.ac.uk/~wbl/aigp3
364. Spector L, Klein J, Keijzer M (2005) The push3 execution stack and the evolution of control. In: Beyer HG, O'Reilly UM, Arnold DV, Banzhaf W, Blum C, Bonabeau EW, Cantu-Paz E, Dasgupta D, Deb K, Foster JA, de Jong ED, Lipson H, Llora X, Mancoridis S, Pelikan M, Raidl GR, Soule T, Tyrrell AM, Watson JP, Zitzler E (eds) GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation, ACM Press, Washington DC, USA, vol 2, pp 1689–1696, URL http://www.cs.bham.ac.uk/~wbl/biblio/gecco2005/docs/p1689.pdf
365. Stender J (ed) (1993) Parallel Genetic Algorithms: Theory and Applications. IOS press
366. Stephens CR, Waelbroeck H (1997) Effective degrees of freedom in genetic algorithms and the block hypothesis. In: Bäck T (ed) Proceedings of the Seventh International Conference on Genetic Algorithms (ICGA97), Morgan Kaufmann, East Lansing, pp 34–40
367. Stephens CR, Waelbroeck H (1999) Schemata evolution and building blocks. Evolutionary Computation 7(2):109–124
368. Sterling T (1998) Beowulf-class clustered computing: Harnessing the power of parallelism in a pile of PCs. In: Koza JR, Banzhaf W, Chellapilla K, Deb K, Dorigo M, Fogel DB, Garzon MH, Goldberg DE, Iba H, Riolo R (eds) Genetic Programming 1998: Proceedings of the Third Annual Conference, Morgan Kaufmann, University of Wisconsin, Madison, Wisconsin, USA, p 883, invited talk
369. Szymanski JJ, Brumby SP, Pope P, Eads D, Esch-Mosher D, Galassi M, Harvey NR, McCulloch HDW, Perkins SJ, Porter R, Theiler J, Young AC, Bloch JJ, David N (2002) Feature extraction from multiple data sources using genetic programming. In: Shen SS, Lewis PE (eds) Algorithms and Technologies for Multispectral, Hyperspectral, and Ultraspectral Imagery VIII, SPIE, vol 4725, pp 338–345, URL http://www.cs.rit.edu/~dre9227/papers/szymanskiSPIE4725.pdf
370. Tackett WA (1993) Genetic generation of "dendritic" trees for image classification. In: Proceedings of WCNN93, IEEE Press, pp IV 646–649, URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/ftp.io.com/papers/GP.feature.discovery.ps.Z
371. Takagi H (2001) Interactive evolutionary computation: Fusion of the capabilities of EC optimization and human evaluation. Proceedings of the IEEE 89(9):1275–1296, invited Paper

372. Tanev I, Uozumi T, Akhmetov D (2004) Component object based single system image for dependable implementation of genetic programming on clusters. Cluster Computing Journal 7(4):347–356, DOI doi:10.1023/B:CLUS.0000039494.39217.c1, URL http://www.kluweronline.com/issn/1386-7857
373. Taylor J, Goodacre R, Wade WG, Rowland JJ, Kell DB (1998) The deconvolution of pyrolysis mass spectra using genetic programming: application to the identification of some eubacterium species. FEMS Microbiology Letters 160:237–246, DOI doi:10.1016/S0378-1097(98)00038-X
374. Teller A (1994) Genetic programming, indexed memory, the halting problem, and other curiosities. In: Proceedings of the 7th annual Florida Artificial Intelligence Research Symposium, IEEE Press, Pensacola, Florida, USA, pp 270–274, URL http://www.cs.cmu.edu/afs/cs/usr/astro/public/papers/Curiosities.ps
375. Teller A (1996) Evolving programmers: The co-evolution of intelligent recombination operators. In: Angeline PJ, Kinnear, Jr KE (eds) Advances in Genetic Programming 2, MIT Press, Cambridge, MA, USA, chap 3, pp 45–68, URL http://www.cs.cmu.edu/afs/cs/usr/astro/public/papers/AiGPII.ps
376. Teller A, Andre D (1997) Automatically choosing the number of fitness cases: The rational allocation of trials. In: Koza JR, Deb K, Dorigo M, Fogel DB, Garzon M, Iba H, Riolo RL (eds) Genetic Programming 1997: Proceedings of the Second Annual Conference, Morgan Kaufmann, Stanford University, CA, USA, pp 321–328, URL http://www.cs.cmu.edu/afs/cs/usr/astro/public/papers/GR.ps
377. Teredesai A, Govindaraju V (2005) GP-based secondary classifiers. Pattern Recognition 38(4):505–512, DOI doi:10.1016/j.patcog.2004.06.010
378. Theiler JP, Harvey NR, Brumby SP, Szymanski JJ, Alferink S, Perkins SJ, Porter RB, Bloch JJ (1999) Evolving retrieval algorithms with a genetic programming scheme. In: Descour MR, Shen SS (eds) Proceedings of SPIE 3753 Imaging Spectrometry V, pp 416–425, URL http://public.lanl.gov/jt/Papers/ga-spie.ps
379. Todd PM, Werner GM (1999) Frankensteinian approaches to evolutionary music composition. In: Griffith N, Todd PM (eds) Musical Networks: Parallel Distributed Perception and Performance, MIT Press, pp 313–340, URL http://www-abc.mpib-berlin.mpg.de/users/ptodd/publications/99evmus/99evmus.pdf
380. Tomassini M, Luthi L, Giacobini M, Langdon WB (2007) The structure of the genetic programming collaboration network. Genetic Programming and Evolvable Machines 8(1):97–103, DOI doi:10.1007/s10710-006-9018-2
381. Trujillo L, Olague G (2006a) Synthesis of interest point detectors through genetic programming. In: Keijzer M, Cattolico M, Arnold D, Babovic V, Blum C, Bosman P, Butz MV, Coello Coello C, Dasgupta D, Ficici SG, Foster J, Hernandez-Aguirre A, Hornby G, Lipson H, McMinn P, Moore J, Raidl G, Rothlauf F, Ryan C, Thierens D (eds) GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation, ACM Press, Seattle, Washington, USA, vol 1, pp 887–894, DOI doi:10.1145/1143997.1144151, URL http://www.cs.bham.ac.uk/~wbl/biblio/gecco2006/docs/p887.pdf
382. Trujillo L, Olague G (2006b) Using evolution to learn how to perform interest point detection. In: et al XYT (ed) ICPR 2006 18th International Conference on Pattern Recognition, IEEE, vol 1, pp 211–214, DOI doi:10.1109/ICPR.2006.1153, URL http://www.genetic-programming.org/hc2006/Olague-Paper-2-ICPR%-2006.pdf

383. Tsang EPK, Li J, Butler JM (1998) EDDIE beats the bookies. Software: Practice and Experience 28(10):1033–1043, DOI doi:10.1002/(SICI)1097-024X(199808)28:10⟨1033::AID-SPE198⟩3.0.CO;2--1, URL http://cswww.essex.ac.uk/CSP/finance/papers/TsBuLi-Eddie-Software98.pdf
384. Turing AM (1948) Intelligent machinery, report for National Physical Laboratory. Reprinted in Ince, D. C. (editor). 1992. Mechanical Intelligence: Collected Works of A. M. Turing. Amsterdam: North Holland. Pages 107127. Also reprinted in Meltzer, B. and Michie, D. (editors). 1969. Machine Intelligence r5. Edinburgh: Edinburgh University Press
385. Turing AM (1950) Computing machinery and intelligence. Mind 49:433–460, URL http://www.cs.umbc.edu/471/papers/turing.pdf
386. Usman I, Khan A, Chamlawi R, Majid A (2007) Image authenticity and perceptual optimization via genetic algorithm and a dependence neighborhood. International Journal of Applied Mathematics and Computer Sciences 4(1):615–620, URL http://www.waset.org/ijamcs/v4/v4-1-7.pdf
387. Vaidyanathan S, Broadhurst DI, Kell DB, Goodacre R (2003) Explanatory optimization of protein mass spectrometry via genetic search. Analytical Chemistry 75(23):6679–6686, DOI doi:10.1021/ac034669a, URL http://dbkgroup.org/Papers/AnalChem75(6679-6686).pdf
388. Venkatraman V, Dalby AR, Yang ZR (2004) Evaluation of mutual information and genetic programming for feature selection in QSAR. Journal of Chemical Information and Modeling 44(5):1686–1692, DOI doi:10.1021/ci049933v
389. Vowk B, Wait AS, Schmidt C (2004) An evolutionary approach generates human competitive coreware programs. In: Bedau M, Husbands P, Hutton T, Kumar S, Sizuki H (eds) Workshop and Tutorial Proceedings Ninth International Conference on the Simulation and Synthesis of Living Systems(Alife XI), Boston, Massachusetts, pp 33–36, artificial Chemistry and its applications workshop
390. Vukusic I, Grellscheid SN, Wiehe T (2007) Applying genetic programming to the prediction of alternative mRNA splice variants. Genomics 89(4):471–479, DOI doi:10.1016/j.ygeno.2007.01.001
391. Walker RL (2001) Search engine case study: searching the web using genetic programming and MPI. Parallel Computing 27(1–2):71–89, URL http://www.sciencedirect.com/science/article/B6V12-42K5HNX-4/1/57eb870c72fb7768bb7d824557444b72
392. Walsh P, Ryan C (1996) Paragen: A novel technique for the autoparallelisation of sequential programs using genetic programming. In: Koza JR, Goldberg DE, Fogel DB, Riolo RL (eds) Genetic Programming 1996: Proceedings of the First Annual Conference, MIT Press, Stanford University, CA, USA, pp 406–409, URL http://cognet.mit.edu/library/books/view?isbn=0262611279
393. Weaver DC (2004) Applying data mining techniques to library design, lead generation and lead optimization. Current Opinion in Chemical Biology 8(3):264–270, DOI doi:10.1016/j.cbpa.2004.04.005, URL http://www.sciencedirect.com/science/article/B6VRX-4CB69R1-2/2/84a354cec9064ed07baab6a07998c942
394. Whigham PA (1995) A schema theorem for context-free grammars. In: 1995 IEEE Conference on Evolutionary Computation, IEEE Press, Perth, Australia, vol 1, pp 178–181
395. Whigham PA (1996) Search bias, language bias, and genetic programming. In: Koza JR, Goldberg DE, Fogel DB, Riolo RL (eds) Genetic Programming 1996:

Proceedings of the First Annual Conference, MIT Press, Stanford University, CA, USA, pp 230–237
396. Whitley D (2001) An overview of evolutionary algorithms: practical issues and common pitfalls. Information and Software Technology 43(14):817–831, DOI doi:10.1016/S0950-5849(01)00188-4, URL http://www.cs.colostate.edu/~genitor/2001/overview.pdf
397. Whitley LD (1994) A Genetic Algorithm Tutorial. Statistics and Computing 4:65–85
398. Willis M, Hiden H, Marenbach P, McKay B, Montague GA (1997a) Genetic programming: An introduction and survey of applications. In: Zalzala A (ed) Second International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications, GALESIA, Institution of Electrical Engineers, University of Strathclyde, Glasgow, UK, URL http://www.staff.ncl.ac.uk/d.p.searson/docs/galesia97surveyofGP.pdf
399. Willis MJ, Hiden HG, Montague GA (1997b) Developing inferential estimation algorithms using genetic programming. In: IFAC/ADCHEM International Symposium on Advanced Control of Chemical Processes, Banaff, Canada, pp 219–224
400. Wilson G, Heywood M (2007) Introducing probabilistic adaptive mapping developmental genetic programming with redundant mappings. Genetic Programming and Evolvable Machines 8(2):187–220, DOI doi:10.1007/s10710-007-9027-9, special issue on developmental systems
401. Wong ML (1998) An adaptive knowledge-acquisition system using generic genetic programming. Expert Systems with Applications 15(1):47–58, URL http://cptra.ln.edu.hk/~mlwong/journal/esa1998.pdf
402. Wong ML (2005) Evolving recursive programs by using adaptive grammar based genetic programming. Genetic Programming and Evolvable Machines 6(4):421–455, DOI doi:10.1007/s10710-005-4805-8, URL http://cptra.ln.edu.hk/~mlwong/journal/gpem2005.pdf
403. Wong ML, Leung KS (1995) Inducing logic programs with genetic algorithms: the genetic logicprogramming system genetic logic programming and applications. IEEE Expert 10(5):68–76, DOI doi:10.1109/64.464935
404. Wong ML, Leung KS (1996) Evolving recursive functions for the even-parity problem using genetic programming. In: Angeline PJ, Kinnear, Jr KE (eds) Advances in Genetic Programming 2, MIT Press, Cambridge, MA, USA, chap 11, pp 221–240
405. Wong ML, Leung KS (2000) Data Mining Using Grammar Based Genetic Programming and Applications, Genetic Programming, vol 3. Kluwer Academic Publishers
406. Wong ML, Wong TT, Fok KL (2005) Parallel evolutionary algorithms on graphics processing unit. In: Corne D, Michalewicz Z, McKay B, Eiben G, Fogel D, Fonseca C, Greenwood G, Raidl G, Tan KC, Zalzala A (eds) Proceedings of the 2005 IEEE Congress on Evolutionary Computation, IEEE Press, Edinburgh, Scotland, UK, vol 3, pp 2286–2293, URL http://ieeexplore.ieee.org/servlet/opac?punumber=10417&isvol=3
407. Woodward AM, Gilbert RJ, Kell DB (1999) Genetic programming as an analytical tool for non-linear dielectric spectroscopy. Bioelectrochemistry and Bioenergetics 48(2):389–396, DOI doi:doi:10.1016/S0302-4598(99)00022-7, URL http://www.sciencedirect.com/science/article/B6TF7-3WJ72RJ-T/2/19fd01a6eb6ae0b8e12b2bb2218fb6e9

408. Wright S (1932) The roles of mutation, inbreeding, crossbreeding and selection in evolution. In: Jones DF (ed) Proceedings of the Sixth International Congress on Genetics, vol 1, pp 356–366
409. Xie H, Zhang M, Andreae P (2006) Genetic programming for automatic stress detection in spoken english. In: Rothlauf F, Branke J, Cagnoni S, Costa E, Cotta C, Drechsler R, Lutton E, Machado P, Moore JH, Romero J, Smith GD, Squillero G, Takagi H (eds) Applications of Evolutionary Computing, EvoWorkshops2006: EvoBIO, EvoCOMNET, EvoHOT, EvoIASP, EvoInteraction, EvoMUSART, EvoSTOC, Springer Verlag, Budapest, LNCS, vol 3907, pp 460–471, DOI doi:10.1007/11732242_41, URL http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=3907&spage=460
410. Yangiya M (1995) Efficient genetic programming based on binary decision diagrams. In: 1995 IEEE Conference on Evolutionary Computation, IEEE Press, Perth, Australia, vol 1, pp 234–239
411. Yu J, Bhanu B (2006) Evolutionary feature synthesis for facial expression recognition. Pattern Recognition Letters 27(11):1289–1298, DOI doi:10.1016/j.patrec.2005.07.026, evolutionary Computer Vision and Image Understanding
412. Yu J, Yu J, Almal AA, Dhanasekaran SM, Ghosh D, Worzel WP, Chinnaiyan AM (2007) Feature selection and molecular classification of cancer using genetic programming. Neoplasia 9(4):292–303, DOI doi:10.1593/neo.07121
413. Yu T (2001) Hierachical processing for evolving recursive and modular programs using higher order functions and lambda abstractions. Genetic Programming and Evolvable Machines 2(4):345–380, DOI doi:10.1023/A:1012926821302
414. Yu T, Chen SH (2004) Using genetic programming with lambda abstraction to find technical trading rules. In: Computing in Economics and Finance, University of Amsterdam
415. Yu T, Riolo RL, Worzel B (eds) (2005) Genetic Programming Theory and Practice III, Genetic Programming, vol 9, Springer, Ann Arbor
416. Zhang BT, Mühlenbein H (1993) Evolving optimal neural networks using genetic algorithms with Occam's razor. Complex Systems 7:199–220, URL http://citeseer.ist.psu.edu/zhang93evolving.html
417. Zhang BT, Mühlenbein H (1995) Balancing accuracy and parsimony in genetic programming. Evolutionary Computation 3(1):17–38, URL http://www.ais.fraunhofer.de/~muehlen/publications/gmd_as_ga-94_09.ps
418. Zhang BT, Ohm P, Mühlenbein H (1997) Evolutionary induction of sparse neural trees. Evolutionary Computation 5(2):213–236, URL http://bi.snu.ac.kr/Publications/Journals/International/EC5-2.ps
419. Zhang M, Smart W (2006) Using gaussian distribution to construct fitness functions in genetic programming for multiclass object classification. Pattern Recognition Letters 27(11):1266–1274, DOI doi:10.1016/j.patrec.2005.07.024, evolutionary Computer Vision and Image Understanding
420. Zhang Y, Rockett PI (2006) Feature extraction using multi-objective genetic programming. In: Jin Y (ed) Multi-Objective Machine Learning, Studies in Computational Intelligence, vol 16, Springer, chap 4, pp 79–106, invited chapter

Resources

Following the publication of [188], the field of GP took off in about 1990 with a period of exponential growth common in the initial stages of successful technologies. Many influential initial papers from that period can be found in the proceedings of the *Intl. Conf. Genetic Algorithms* (ICGA-93, ICGA-95), the *IEEE Confs. on Evolutionary Computation* (EC-1994), and the *Evolutionary Programming Conference*. A surprisingly large number of these are now available online. After almost twenty years, GP has matured and is used in a wondrous array of applications. From banking [265] to betting [383], from bomb detection [102] to architecture [282], from the steel industry to the environment [157], from space [234] to biology [159], and many others (as we have seen in Sect. 7). In 1996 it was possible to list (almost all) GP applications [201], but today the range is far too great, so here we simply list some GP resources, which, we hope, will guide readers towards their goals.

1 Key Books

There are today more than 31 books written in English principally on GP or its applications, with more being written. These start with John Koza's 1992 *Genetic Programming* (often referred to as 'Jaws'). Koza has published four 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 analogue circuits; *Genetic Programming 4* (2003) uses GP for automatic invention.

MIT Press published three volumes in the series *Advances in Genetic Programming* (1994, 1996, 1999).

The joint GP/genetic algorithms Kluwer book series edited by Koza and Goldberg now contains 14 books, starting with *Genetic Programming and*

Data Structures [203]. Apart from 'Jaws', these tend to be for the GP specialist.

1997 saw the introduction of the first textbook dedicated to GP [25].

Eiben [87] and Goldberg [115] provide general treatment on evolutionary algorithms.

Other titles include: *Principia Evolvica – Simulierte Evolution mit Mathematica* (in German) [154] (English version [156]), *Data Mining Using Grammar Based Genetic Programming and Applications* [405], *Genetic Programming* (in Japanese) [151], and *Humanoider: Sjavlarande robotar och artificiell intelligens* (in Swedish) [274].

Readers interested in mathematical and empirical analyses of GP behavior may find *Foundations of Genetic Programming* [222] useful.

2 Videos

Each of Koza's four books has an accompanying illustrative video. These are now available in DVD format. Furthermore, a small set of videos on specific GP techniques and applications is available from `Google Video` and `YouTube`.

3 Key Journals

In addition to GP's own *Genetic Programming and Evolvable Machines* journal (Kluwer), *Evolutionary Computation*, the *IEEE Trans. Evolutionary Computation*, *Complex Systems* (Complex Systems Publication, Inc.), and many others publish GP articles. The *GP bibliography* (http://www.cs.bham.ac.uk/~wbl/biblio/) lists a further 375 different journals worldwide that have published articles related to GP.

4 Key International Conferences/Workshops

EuroGP has been held every year since 1998. All *EuroGP* papers are available on line as part of Springer's LNCS series. The original annual *Genetic Programming* conference was hosted by Koza in 1996 at Stanford. Since 1999 it has been combined with the *Intl. Conf. Genetic Algorithms* to form *GECCO*; 98% of *GECCO* papers are available online. The Michigan-based *Genetic Programming Theory and Practice Workshop* [284,322,323,415] will shortly publish its fifth proceedings [324]. Other EC conferences, such as *CEC*, *PPSN*, *Evolution Artificielle*, and *WSC*, also regularly contain GP papers.

5 Online Resources

One of the reasons behind the success of GP is that it is easy to implement your own version. People have coded GP in a huge range of different languages, such as Lisp, C, C++, Java, JavaScript, Perl, Prolog, Mathematica, Pop-11, MATLAB, Fortran, Occam and Haskell. Typically these evolve code which looks like a very cut down version of Lisp. However, admirers of grammars claim the evolved language can be arbitrarily complex, and certainly programs in functional and other high level languages have been automatically evolved. Conversely, many successful programs in machine code or low-level languages have also climbed from the primordial ooze of initial randomness.

Many GP implementations can be freely downloaded. Two that have been available for a long time and remain popular are: Sean Luke's `ECJ` (in Java), and Douglas Zonger's 'little GP' `lilGP` (in C). A number of older (unsupported) tools can be found at ftp://cs.ucl.ac.uk/genetic/ftp.io.com/. The most prominent commercial implementation remains `Discipulus` [99].

There is a lot of information available on the the world wide web, although, unfortunately, Internet addresses (URLs) change rapidly. Therefore we simply name useful pages here (rather than give their URL). A web search will usually quickly locate them.

At the time of writing, the *GP bibliography* contains about 5000 GP entries. About half the entries can be downloaded immediately. There are a variety of interfaces including a graphical representation of GP's collaborative network (see Fig. 1). The HTML pages are perhaps the easiest to use. They allow quick jumps between papers linked by authors, show paper concentrations and in many cases direct paper downloads. The collection of computer sciences bibliographies provides a comprehensive Lucene syntax search engine. Bibtex and Refer files can also be searched but are primarily intended for direct inclusion of bibliographic references in papers written in LaTeX and Microsoft Word, respectively.

Almost since the beginning there has been an open active email discussion list: the *GP discussion group*, which is hosted by `Yahoo!` For more reflective discussions, the *EC-Digest* comes out once a fortnight and often contains GP related announcements, while the organization behind *GECCO* also runs a quarterly *SIGEvolution* newsletter.

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', as well as LISP code for implementing GP, as in [188].

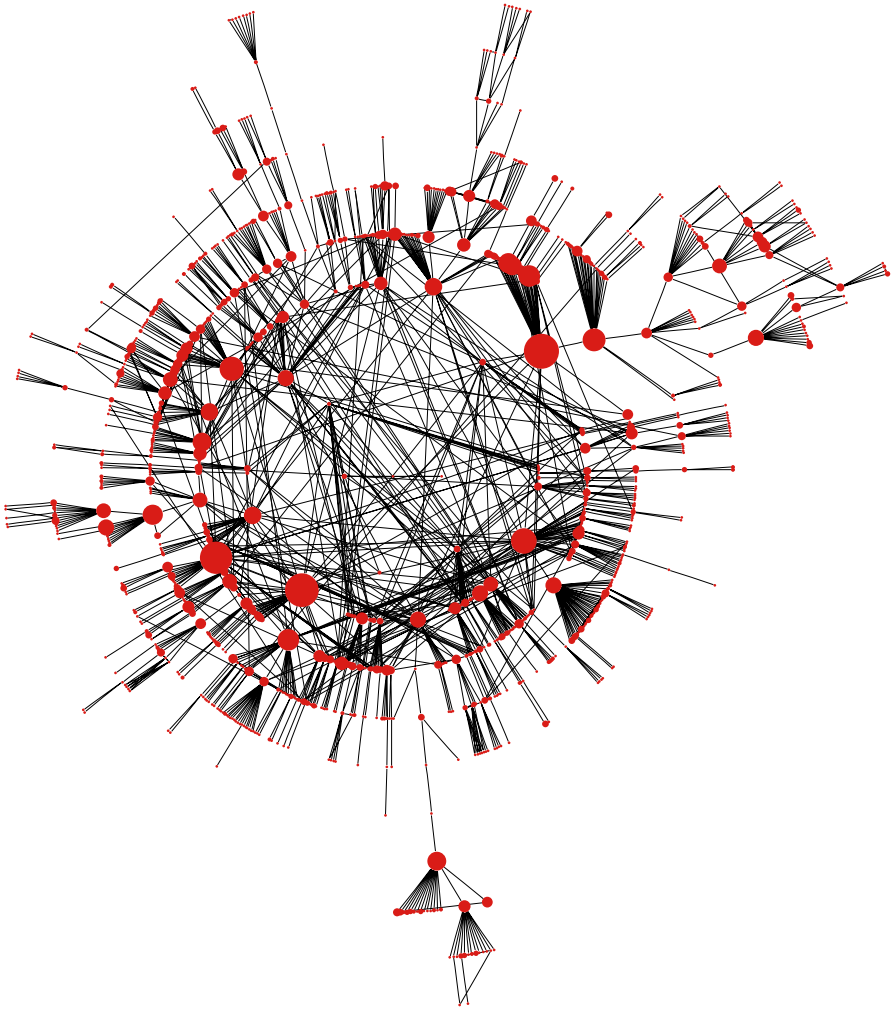

Fig. 1. Co-author connections within GP. Each of the 1141 dots indicates an author. The lines link people who have co-authored one or more papers. (To reduce clutter only links to first authors are shown.) The online version is annotated by JavaScript and contains hyperlinks to authors and their GP papers. The graph was created by `GraphViz twopi`, which tries to place strongly connected people close together. It is the 'centrally connected component' [380] and contains approximately half of all GP papers. The remaining papers are not linked by co-authorship to this graph. Several of the larger unconnected graphs are also available online via the *gp-bibliography* web pages