Genetic Programming
An Introductory Tutorial and a Survey of Techniques and Applications

Riccardo Poli
Department of Computing and Electronic Systems
University of Essex, UK
rpoli@essex.ac.uk

William B. Langdon
Departments of Biological and Mathematical Sciences
University of Essex, UK
wlangdon@essex.ac.uk

Nicholas F. McPhee
Division of Science and Mathematics
University of Minnesota, Morris, USA
mcphee@morris.umn.edu

John R. Koza
Stanford University, Stanford, California
john@johnkoza.com

Technical Report CES-475
ISSN: 1744-8050
October 2007

Abstract

This paper introduces genetic programming (GP) – a set of evolutionary computation techniques for getting computers to automatically solve problems without having to tell them explicitly how to do it. Since its inception, GP has been used to solve many practical problems, producing a number of human competitive results and even patentable new inventions. We start with a gentle introduction to the basic representation, initialisation and operators used in GP, complemented by a step by step description of their use for the solution of an illustrative problem. We then progress to discuss a variety of alternative representations for programs and more advance specialisations of GP. A multiplicity of real-world applications of GP are then presented to illustrate the scope of the technique. For the benefits of more advanced readers, this is followed by a series of recommendations and suggestions to obtain the most from a GP system. Although the paper has been written with beginners and practitioners in mind, for completeness we also provide an overview of the theoretical results and models available to date for GP. The paper is concluded by an appendix which provides a plethora of pointers to resources and further reading.

Contents

1 Introduction **5**
- 1.1 GP in a Nutshell . . . 5
- 1.2 Overview of the Paper . . . 6

2 Representation, Initialisation and Operators in Tree-based GP **7**
- 2.1 Representation . . . 7
- 2.2 Initialising the Population . . . 9
- 2.3 Selection . . . 11
- 2.4 Recombination and Mutation . . . 11

3 Getting Ready to Run Genetic Programming **14**
- 3.1 Steps 1: Terminal Set . . . 14
- 3.2 Step 2: Function Set . . . 15
  - 3.2.1 Closure . . . 15
  - 3.2.2 Sufficiency . . . 17
  - 3.2.3 Evolving Structures other than Programs . . . 17
- 3.3 Step 3: Fitness Function . . . 18
- 3.4 Steps 4 and 5: Parameters and Termination . . . 19

4 Example of a Run of Genetic Programming **20**
- 4.1 Preparatory Steps . . . 20
- 4.2 Step-by-Step Sample Run . . . 22
  - 4.2.1 Initialisation . . . 22
  - 4.2.2 Fitness Evaluation . . . 22
  - 4.2.3 Selection, Crossover and Mutation . . . 23

5 Advanced Tree-based GP Techniques **25**
- 5.1 Automatically Defined Functions . . . 25
- 5.2 Program Architecture and Architecture-Altering Operations . . . 26
- 5.3 Genetic Programming Problem Solver . . . 26
- 5.4 Constraining Syntactic Structures . . . 27
  - 5.4.1 Enforcing Particular Structures . . . 27
  - 5.4.2 Strongly Typed GP . . . 28
  - 5.4.3 Grammar Based Constraints . . . 28
  - 5.4.4 A Cautionary Note . . . 29
- 5.5 Developmental Genetic Programming . . . 30

6 Linear and Graph-based GP **31**
- 6.1 Linear Genetic Programming . . . 31
- 6.2 Graph Based Genetic Programming . . . 33

7 Applications **34**
7.1 Curve Fitting, Data Modelling, and Symbolic Regression 34
7.2 Human Competitive Results – *the Humies* 37
7.3 Image and Signal Processing 40
7.4 Financial Trading, Time Series Prediction and Economic Modelling 42
7.5 Industrial Process Control 43
7.6 Medicine, Biology and Bioinformatics 43
7.7 Mixing GP with Other Techniques 44
7.8 GP to Create Searchers and Solvers – Hyper-heuristics 44
7.9 Artistic 45
7.10 Entertainment and Computer Games 45
7.11 Where can we Expect GP to Do Well? 46

8 Tricks of the Trade **46**
8.1 Getting Started 46
8.2 Presenting Results 47
8.3 Reducing Fitness Evaluations/Increasing their Effectiveness 48
8.4 Co-evolution 50
8.5 Reducing Cost of Fitness with Caches 50
8.6 GP Running in Parallel 52
8.6.1 Master-slave GP 52
8.6.2 Geographically Distributed GP 53
8.6.3 GP Running on GPUs 55
8.7 GP Trouble-shooting 56

9 Genetic Programming Theory **57**
9.1 Mathematical Models 57
9.2 Search Spaces 58
9.3 Bloat 60

10 Conclusions **61**

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" [415]. Machine learning pioneer Arthur Samuel, in his 1983 talk entitled "AI: Where It Has Been and Where It Is Going" [366], stated that the main goal of the fields of machine learning and artificial intelligence is:

> "to get machines to exhibit behaviour, 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 paper is intended to give an overview of the basics of GP, to summarise 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 paper. Thus this paper 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 Figure 1. During the process, GP constructs new programs by applying genetic operations which are specialised 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:

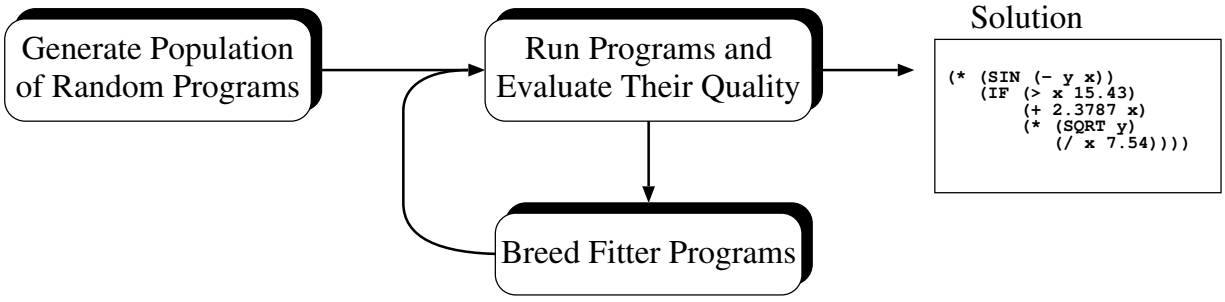

Figure 1: GP's main loop.

Algorithm 1 Abstract GP algorithm.

1: Randomly create an *initial population* of programs from the available primitives (see Section 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 Section 2.3).
5: Create new individual program(s) by applying *genetic operations* with specified probabilities (see Section 2.4).
6: **until** an acceptable solution is found or some other stopping condition is met (e.g., reaching a maximum number of generations).
7: **return** the best-so-far individual.

- **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, e.g., Section 5.1), and some of these then include *architecture-altering operations* which randomly alter the architecture (e.g., 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 Paper

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

This is followed by descriptions of some more advanced GP techniques including: automatically defined functions (Section 5.1) and architecture-altering operations (Section 5.2), the GP problem solver (Section 5.3), systems that constrain the syntax of evolved programs in some way (e.g., using grammars or type systems; Section 5.4) and developmental GP (Section 5.5). Alternative program representations, namely linear GP (Section 6.1) and graph-based GP (Section 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 modelling (Section 7.1), human competitive results (Section 7.2) and much more, and a substantial collection of "tricks of the trade" used by experienced GP practitioners (Section 8). We also give an overview of some of the considerable work that has been done on the theory of GP (Section 9).

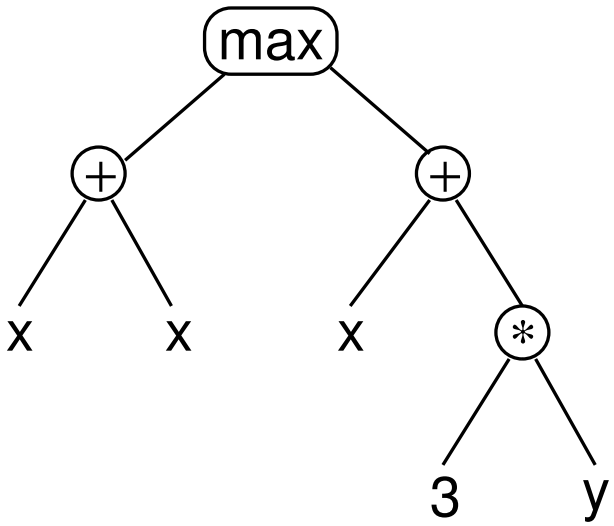

Figure 2: GP syntax tree representing `max(x*x,x+3y)`.

After concluding the paper (Section 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, Initialisation 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 (Section 2.1), how one might construct the initial, random population (Section 2.2), and how selection (Section 2.3) as well as recombination and mutation (Section 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 (`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 (e.g., 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 Figure 3. We will call these (sub)trees *branches*. The number and type of the branches in

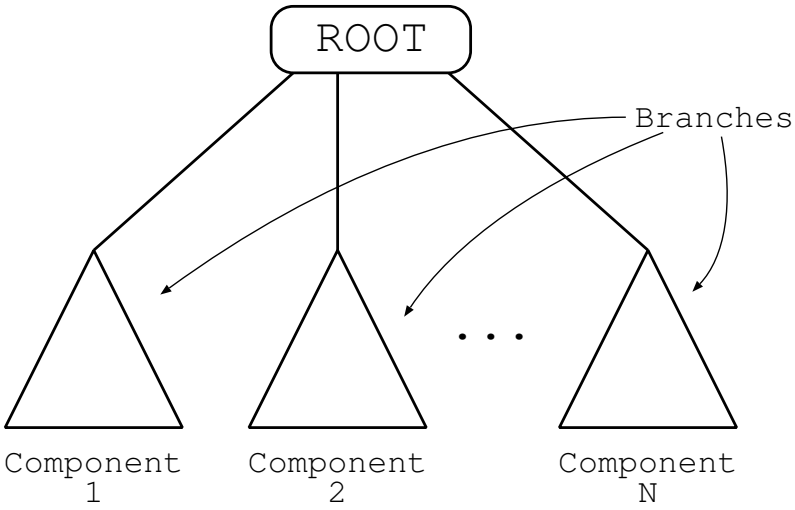

Figure 3: Multi-component program representation.

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 (e.g., Lisp and Prolog), many recent languages (e.g., Ruby and Python), and the languages associated with several scientific programming tools (e.g., 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 (e.g., it is sometimes useful to attach additional information to nodes, which may require that they be explicitly represented). There are also numerous

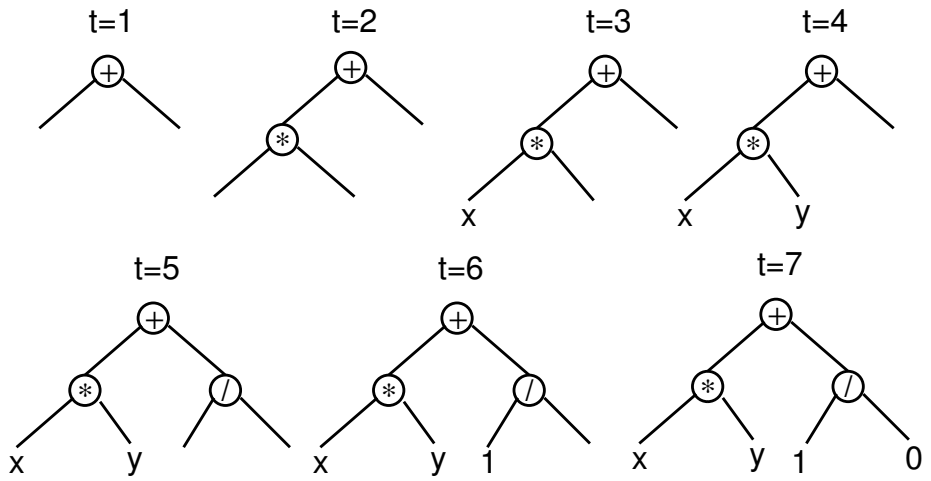

Figure 4: Creation of a full tree having maximum depth 2 (and therefore a total of seven nodes) using the Full initialisation method (t=time).

high-quality, freely available GP implementations (see the resources in the appendix at the end of this paper for more information).

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

2.2 Initialising 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, 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

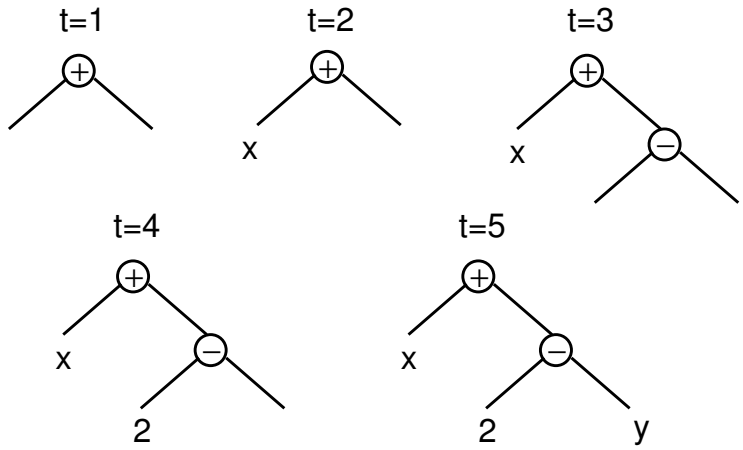

Figure 5: Creation of a five node tree using the Grow initialisation method with a maximum depth of 2 (t=time). A terminal is chosen at $t = 2$, causing the left branch of the root to be closed at that point even though the maximum depth had not been reached.

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.

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 [203] proposed a combination called *ramped half-and-half*. 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 initialisation mechanisms, however, have been developed (e.g., [263]) 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.

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** $\left(\text{method} = \text{grow}\ \textbf{and}\ \text{rand}() < \frac{|\text{term\_set}|}{|\text{term\_set}|+|\text{func\_set}|}\right)$ **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: **end for**
8: expr = (func, arg_1, arg_2, ...);
9: **end if**
10: **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.

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 (e.g., [313, 374]), demes [242] (see Section 8.6), fitness sharing [125] 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 many times in the EA literature, 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 Figure 6.

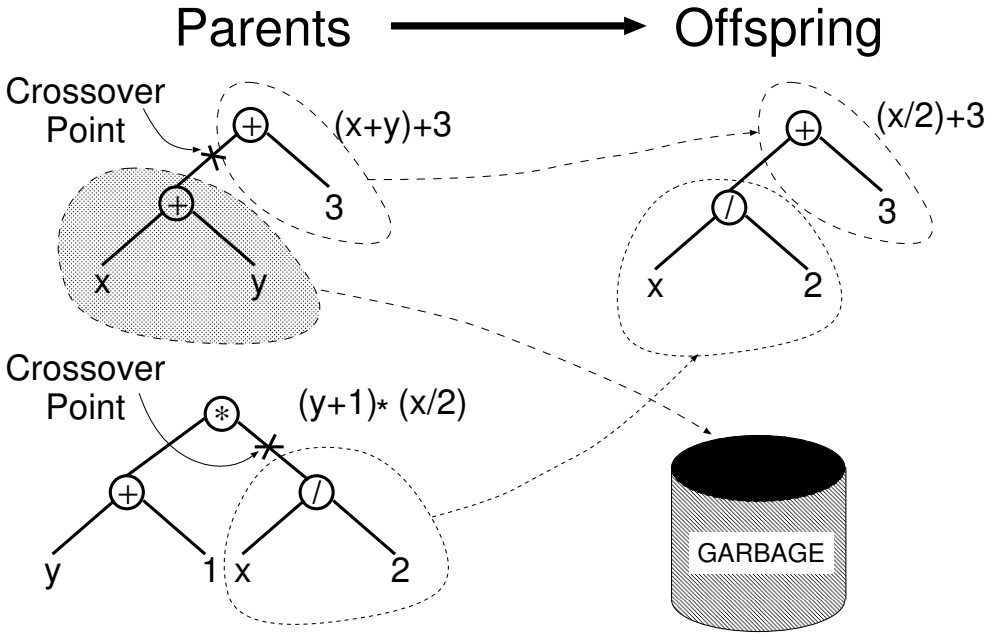

Figure 6: Example of subtree crossover.

Except in technical studies on the behaviour 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 2, 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 (i.e., 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* [326, 329, 239] 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 analyses 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 (i.e. the same arity in the nodes encountered traversing the trees from the root node). In *context-preserving crossover* [87], 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 (i.e., they are not limited to the common region). In *size-fair crossover* [224, 243] 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

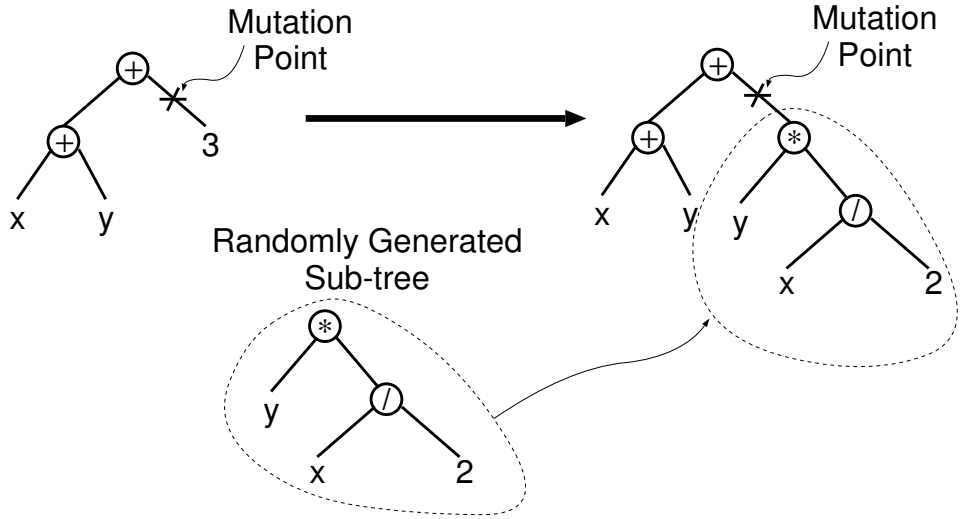

Figure 7: Example of subtree mutation.

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 [328].

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 Figure 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 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. [371] mutates constants by adding Gaussianly distributed random noise to them. However, others use a variety of potentially expensive optimisation tools to try and fine tune an existing program by finding the "best" value for constants within it. E.g., [369] uses "a numerical partial gradient ascent ... to reach the nearest local optimum" to modify all constants in a program, while [378] uses simulated annealing to stochastically update numerical values within individuals.

While mutation is not necessary for GP to solve many problems, [307] 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 modelling 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 Steps 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, i.e., 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 (e.g., `x`, `y`);
- *functions with no arguments*, which are, therefore, interesting either because they return different values in different invocations (e.g., 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 (e.g., `go_left()`); and
- *constants*, which can either be pre-specified or randomly generated as part of the tree creation process.

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`

Note that using a primitive such as `rand` can cause the behaviour 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 initialising 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 Section 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`.

3.2.1 Closure

For GP to work effectively, most function sets are required to have an important property known as *closure* [203], 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 Section 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 (e.g., `+`, `-`, `*`, `/`), 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, e.g., 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 Section 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 behaviour 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

[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.

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 (e.g., 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.

3.2.2 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, i.e., that the set of all the possible recursive compositions of such primitives includes at least one 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, e.g., $\exp(x)$, 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.

3.2.3 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, etc.). 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, etc. This is further discussed in Section 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 (i.e., 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 synthesise an amplifier, the fitness function is the mechanism for telling GP to synthesise 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, etc.) required to bring a system to a desired *target state*, the *accuracy* of the program in recognising patterns or classifying objects into classes, the *payoff* that a game-playing program produces, the *compliance* of a structure with user-specified design criteria, etc.

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 Figure 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.

In some problems we are interested in the *output* produced by a program, i.e., 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, i.e., functions that do more than just return a value, but, for example, 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

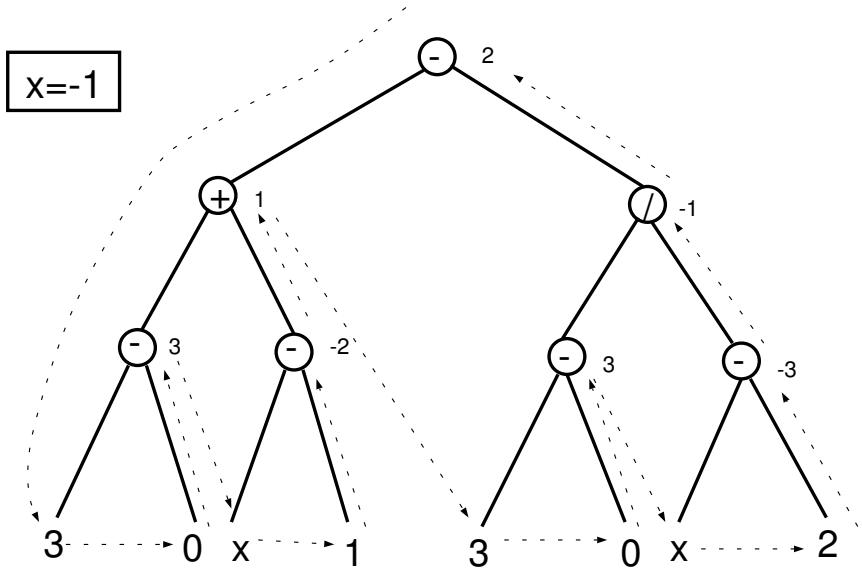

Figure 8: Example interpretation of a syntax tree (the terminal x is a variable has a value of -1). The number to the right of each internal node represents the result of evaluating the subtree root at that node.

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,*, i.e., 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 [80], 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.

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:    end if
8: else
9:    if expr is a variable or expr is a constant then
10:      value = expr {Terminal variable or constant: just read the value}
11:   else
12:      value = expr() {Terminal 0-arity function: execute}
13:   end if
14: end if
15: 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, etc.

4 Example of a Run of Genetic Programming

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 Section 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 Section 3.1. Thus the terminal set, T, is

$$T = \{x, \Re\}.$$

The statement of the problem is somewhat flexible in that it does not specify what func-

tions 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

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

where % is protected division as discussed in Section 3.2.1.

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 Figure 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 Section 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.

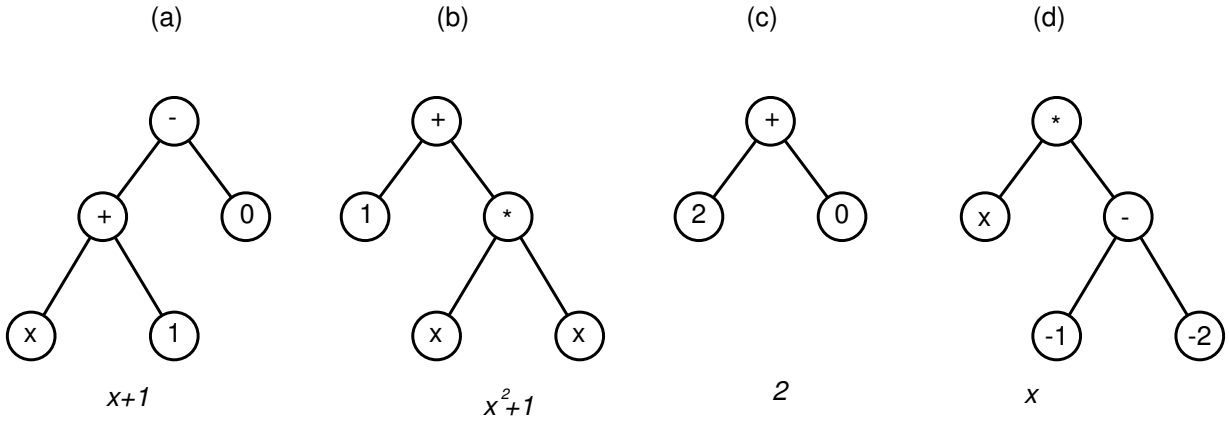

Figure 9: Initial population of four randomly created individuals of generation 0

4.2 Step-by-Step Sample Run

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

4.2.1 Initialisation

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

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

4.2.2 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 Figure 9 produce outputs that deviate by different amounts from the target function x^2+x+1. Figure 10 compares the plots of each of the four individuals in Figure 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 (Figure 10a). The sum of absolute errors for the parabola x^2+1 (the second individual) is 11.0 (Figure 10b). The sums of the absolute errors for the remaining two individuals are 17.98 (Figure 10c) and 28.7 (Figure 10d), respectively.

As can be seen in Figure 10, the straight line x+1 (Figure 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

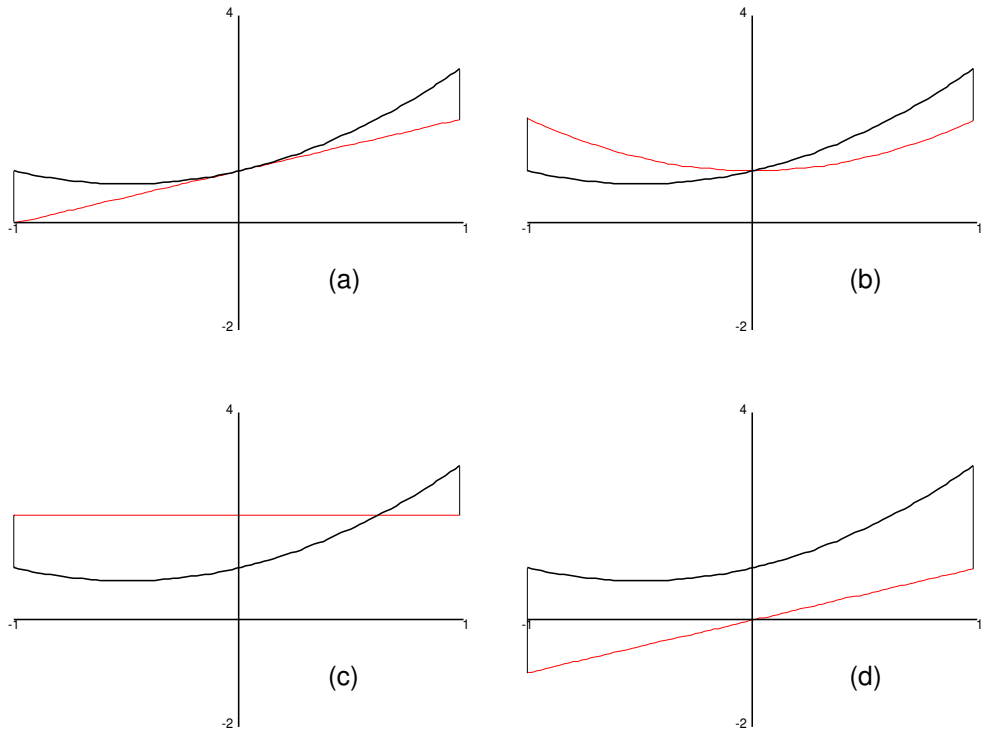

Figure 10: Graphs of the evolved functions from generation 0. The heavy line in each plot is the target function $x^2 + x + 1$, with the other line being the evolved functions from the first generation (see Figure 9). The fitness of each of the four randomly created individuals of generation 0 is approximately 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.

blind (and very limited) random search of generation 0. In the valley of the blind, the one-eyed man is king.

4.2.3 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 (Figure 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

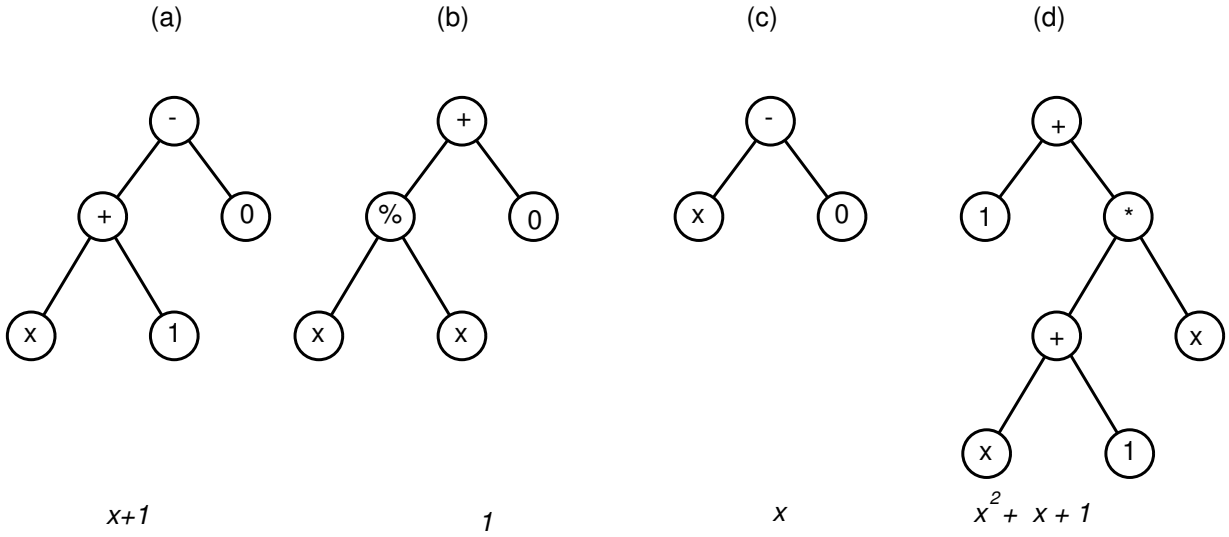

Figure 11: Population of generation 1 (after one reproduction, one mutation, and one two-offspring crossover operation).

next generation (generation 1). It is shown in Figure 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 (Figure 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 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 Figure 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 Figures 9a and the below-average tree in Figure 9d. One point of the first parent, namely the + function in Figure 9a, is randomly picked as the crossover point for the first parent. One point of the second parent, namely the leftmost terminal x in Figure 9d, is randomly picked as the crossover point for the second parent. The crossover operation is then performed on the two parents. The offspring (Figure 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 Figure 9b as the first parent, and the individual in Figure 9a as the second. Let us further imagine that crossover picks the leftmost terminal x in Figure 9b as a crossover point for the first parent, and the + function in Figure 9a as the crossover point for the second parent. Now the offspring (Figure 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 (Figure 11d) is then designated as the result of the run.

Note that the best-of-run individual (Figure 11d) incorporates a good trait (the quadratic term x^2) from the first parent (Figure 9b) with two other good traits (the linear term x and constant term of 1) from the second parent (Figure 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 organise 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 organise these components into hierarchies in which top level components call lower level ones, which call still lower levels, etc.

While several different mechanisms for evolving reusable components have been proposed (e.g., [13, 359]), Koza's *Automatically Defined Functions* (ADFs) [204] have been the most successful way of evolving reusable components.

When ADFs are used, a program consists of one (or more) function-defining trees (i.e., ADFs) as well as one or more main result-producing trees (see Figure 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 *Genetic Programming II* [204] and the videotape [205].

Koza proposed also 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 *Genetic Programming III* [211].

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 (e.g., 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, i.e., perform an architecture-defining preparatory step in addition to the five itemised in Section 2.

2. The run may employ evolutionary selection of the architecture (as described in [204]), 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 [204]) 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 *Genetic Programming III* [211]. 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, cf. Section 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 Section 5.2 to dynamically create, duplicate, and delete subroutines and loops during the run of GP. Additionally, in version 2.0 of GPPS [211, 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 Section 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, etc., 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 analysed, 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 analyse [147], [242, p126].

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.

5.4.1 Enforcing Particular Structures

If a particular structure is believed or known to be important then one can modify the GP system to require that all individuals to have that structure [203]. 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. (I.e., 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 [242, p126].

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.

5.4.2 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 [285]. 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 Section 3, we might have a domain with both numeric and Boolean terminals (e.g., `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 [285], multi-level type systems [148], and fully polymorphic, higher-order type systems with generics [447].

5.4.3 Grammar Based Constraints

Another natural way to express constraints is via grammars, and these have been used in GP in a variety of ways [426, 134, 436, 305, 153]. Many of these simply use a grammar as a means of expressing the kinds of constraints discussed above in Section 5.4.1. One could enforce the structure for the period function using a grammar such as the following:

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

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 [153].

Another major area is grammatical evolution (GE) [363, 305]. 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):

$$
\begin{array}{ll}
 & \underline{\textit{tree}} \\
\rightarrow & \quad \langle\ 39 \bmod 1 = 0, \text{i.e., there is only one option}\ \rangle \\
 & \underline{E} \times \sin(E \times t) \\
\rightarrow & \quad \langle\ 7 \bmod 2 = 1, \text{i.e., choose second option}\ \rangle \\
 & (\underline{E}\ \texttt{op}\ E) \times \sin(E \times t) \\
\rightarrow & \quad \langle\ 2 \bmod 2 = 0, \text{i.e., take the first option}\ \rangle \\
 & (\underline{\texttt{const}}\ \texttt{op}\ E) \times \sin(E \times t) \\
\rightarrow & \quad \langle\ 83 \bmod 3 = 2, \text{again, only one option, generate an ephemeral constant}\ \rangle \\
 & (z\ \underline{\texttt{op}}\ E) \times \sin(E \times t) \\
\rightarrow & \quad \langle\ 66 \bmod 4 = 2, \text{take the third option}\ \rangle \\
 & (z\ \times\ \underline{E}) \times \sin(E \times t) \\
\ldots & \\
 & (z\ \times\ x) \times \sin(z \times t)
\end{array}
$$

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 [305] for further details on this and other aspects of grammatical evolution.

5.4.4 A Cautionary Note

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 comes at a price. An expressive type system typically requires more complex machinery to support it. It also 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 [285, 447, 347, 370].

Also, while the type system may constrain the search space, it is not guaranteed that this will make the evolutionary search process easier. There is no promise, for example, that the type system will *significantly* increase the density of solutions or (perhaps more importantly) approximate solutions. It is also possible that introducing the type system might 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 [239] to the landscapes generated by typed systems in order to better understand the impact of these constraints on the underlying search spaces.

In an extreme case again, a constraint system could generate a needle-in-the-haystack situation, where the search space is indeed much smaller, but there is no smooth path to the solution. It might then be preferable to have a transformation that actually increases the size of the search space, but also generates a gentle slope to the a solution. So while types and other constraint systems can be powerful tools, one 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 [132, 133, 131] and electronic circuits [214, 212, 211], 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.

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 [314, 306, 24]? 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 [298, 300, 72, 97].

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, cf. Figure 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, Figure 13 shows homologous crossover. Many other crossover and mutation operations are possible [244].

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, Peter Nordin started by evolving machine code for SUN computers [298]. Ron Crepeau [72] targeted the Z80. Kwong Sak Leung's linear GP [249] 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 [299] wrapped each machine code GP individual inside a C function. 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

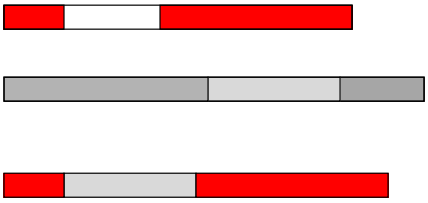

Figure 12: Typical linear GP crossover. Two instructions are randomly chosen in each parent (top two genomes) as cut points. If the code fragment excised from the first parent is replaced with the code fragment excised form the second to give the child (lower chromosome).

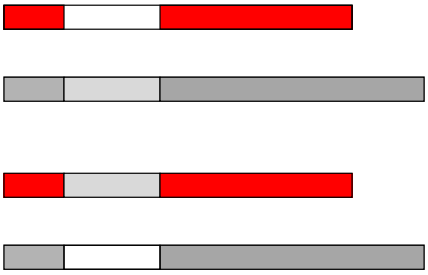

Figure 13: Discipulus' "homologous" crossover [110, 300, 108]. Two parents (top two programs) crossover to yield two child programs (bottom). The two crossover cut points are the same in both parents. Note code does not change its position relative to the start of the program (left edge) and the child programs are the same lengths as their parents. Homologous crossover is often combined with a small amount of normal two point crossover (Figure 12) to introduce length changes into the GP population.

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 (e.g. 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 [36]. In some ways the search space of linear GP is easier to analyse than that of trees [223, 225, 226, 227, 244]. For example, we have used the T7 and T8 architectures (in simulation) for several large scale experimental and mathematical analysis of Turing complete GP [238, 247, 232, 246]. For these reasons, it makes sense to consider linear "machine" code GP, for example, in Java. Since Java is usually run on a virtual machine, almost by definition this requires a virtual machine (like Leung [249]) to interpret the evolved byte code [264, 143].

Output R0..R7	Arg 1 R0..R7	Opcode + − * /	Arg 2 0...127 or R0..R7

Figure 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.

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 [108], which has been used from applications ranging from Bioinformatics [421] to robotics [245] and bomb disposal [85].

Generally, in linear GP instructions take the form shown in Figure 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) [317, 319]. 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 unlabelled, in which case PDGP can be considered a generalisation 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 [405] used a combination of GP and linear discrimination to obtain parallel classification programs for signals and images. 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 [281, 282], 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, in [30], 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 (prioritisation) for actions with side effects. Andre, Bennet and Koza [6] 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 [423] 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 (Sections 7.1– 7.10), devoting particular attention to the important areas of symbolic regression (Section 7.1) and human-competitive results (Section 7.2). We conclude the section with guidelines for the choice of application areas (Section 7.11).

7.1 Curve Fitting, Data Modelling, 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 behaviour, often in

a very application-dependent manner. In many problems, however, the goal is *finding a function* whose output has some desired property, e.g., it matches some target values (as in the example given in Section 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 there is a strong 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 [203], and continues to be a widely studied domain [49, 250, 136, 177].

The steps necessary to solve symbolic regression problems include the five preparatory steps mentioned in Section 2. However, while in the example in Section 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* [171]. 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, i.e., the thing that we want to predict (e.g., the soft sensor value), and which other quantities are the independent variables, i.e., the information we want to use to make the prediction (e.g., 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, i.e., we need to decide which independent variables to keep and which to leave out.

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). Cf. Figure 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 total of 691 lines									

There are problems where more than one output (prediction) is required. For example, Table 2 shows a dataset 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 [245] (the robot is shown in Figure 15 during the acquisition of a data sample). In situations like this, one can use GP individuals including multiple trees (as in Figure 3), graph-based GP with multiple output nodes (see Section 6.2), linear GP with multiple output registers (see Section 6.1), a single GP tree with primitives operating on vectors, etc.

Once a suitable dataset 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? etc.) 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 (e.g., the data is known to be periodic, so the function should probably be based on a something like sin), then applying some sort of constraint, like those discussed in Section 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

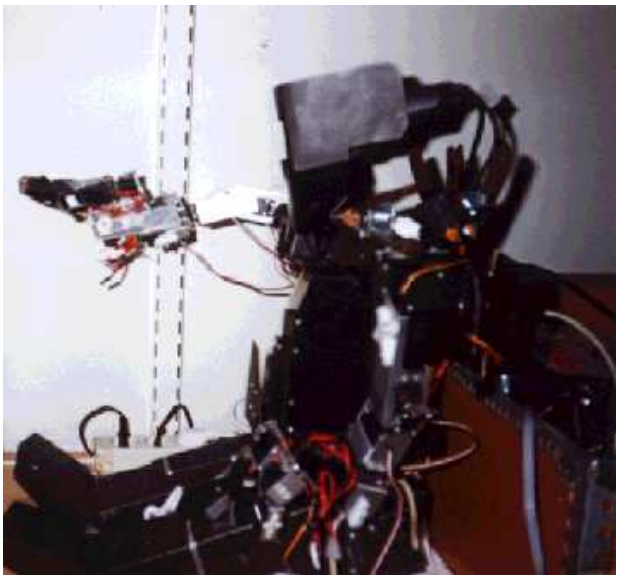

Figure 15: Elvis sitting with right hand outstretched. The apparent position and size of the bright red laser attached to his finger tip is recorded (see Table 2). The data are then used to train a GP to move the robot's arm to a spot in three dimensions using only its eyes.

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 dataset, as we did in Section 4.2.2.

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 behaviour objectively. This led him to propose an imitation game, now known as the Turing test [416]. Unfortunately, the Turing test is not usable in practice, and so, there is a need for more workable objective tests of machine intelligence.

Koza [215] recently proposed to shift the attention from the notion of intelligence to the notion of *human competitiveness*. A result cannot acquire the rating of "human competitive" merely because it is endorsed by researchers *inside* the specialised 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 recognised 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 [219] 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 [390, 391].

- Creation of a competitive soccer-playing program for the RoboCup 1997 competition [262].

- Creation of algorithms for the transmembrane segment identification problem for proteins [204, Sections 18.8 and 18.10] and [211, Sections 16.5 and 17.2].

- Creation of a sorting network for seven items using only 16 steps [211, Sections 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 [211, Section 45.3]; circuits for squaring, cubing, square root, cube root, logarithm, and Gaussian functions [211, Section 47.5.3]; a circuit for time-optimal control of a robot [211, Section 48.3]; an electronic thermometer [211, Section 49.3]; a voltage-current conversion circuit [218, Section 15.4.4].

- Creation of a cellular automata rule for the majority classification problem that is better than all known rules written by humans [6].

- Synthesis of topology for controllers, including: a PID (proportional, integrative, and derivative) [218, Section 9.2] and a PID-D2 (proportional, integrative, derivative, and second derivative) [218, Section 3.7] controllers; PID tuning rules that outperform the Ziegler-Nichols and Astrom-Hagglund tuning rules [218, Chapter 12]; three non-PID controllers that outperform a PID controller that uses the Ziegler-Nichols or Astrom-Hagglund tuning rules [218, Chapter 13].

In total [219] 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 15 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 general-purpose controllers that outperform controllers employing tuning rules that have been in widespread use in industry for most of the 20^{th} 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 prize ($10 000) 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 Figure 16) [256] and for evolutionary quantum computer programming [387]. There were 3 silver medals in 2004: one for evolving local search heuristics for SAT using GP [114], one for the application of GP to the synthesis of complex kinematic mechanisms [254], and one for organisation design optimisation using GP [189, 190]. Also, four of the 2005 medals were awarded for GP applications: the invention of optical lens systems [1, 209], the evolution of quantum Fourier transform algorithm [271], evolving assembly programs for Core War [68], and various high-performance game players for Backgammon, Robocode and Chess endgame [17, 16, 145, 380]. In 2006 again GP scored a gold medal with the synthesis of interest point detectors for image analysis [412, 413], while it scored a silver medal in 2007 with the evolution of an efficient search algorithm for the Mate-in-N problem in Chess [146] (see Figure 17).

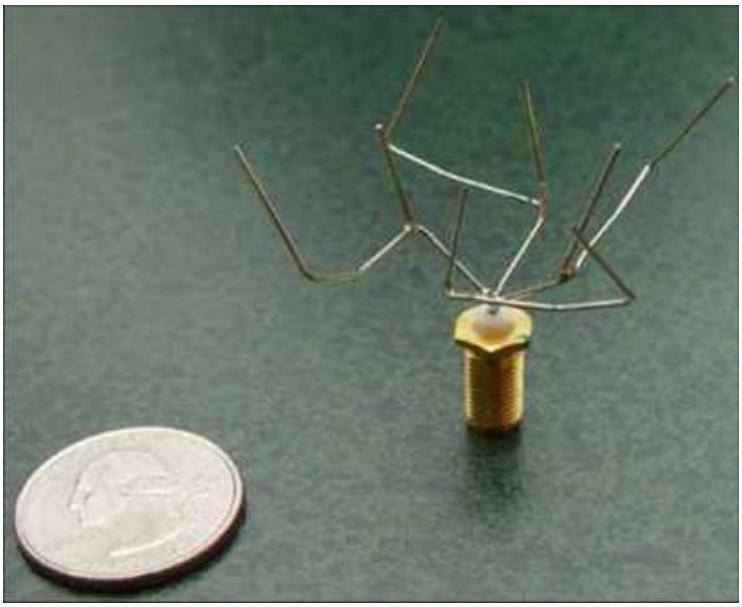

Figure 16: Award winning human-competitive antenna design produced by GP.

Note that many human competitive results were presented at the Humies 2004–2007 competitions (e.g., 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

Rich 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 [137]. Several applications of GP for image processing have been for military uses. E.g. Walter Tackett evolved algorithms to find tanks in infrared images [400]. Daniel Howard evolved program to pick out ships from SAR radar mounted on satellites in space [158] and to locate ground vehicles from airborne photo reconnaissance [159]. He also used GP to process surveillance data for civilian purposes. E.g. to predict motorway traffic jams from subsurface traffic speed measurements [157]. Using to satellite SAR radar, Jason Daida's team evolved algorithms to find features in polar sea ice [74]. Optical satellite images can also be used for environmental studies [52] and for prospecting for valuable minerals [360]. Anna Esparcia Alcazar used GP to find recurrent filters (including artificial neural networks ANN [101]) for one dimensional electronic signals [377]. Local search (simulated annealing or gradient descent) can be used to adjust or fine-tune "constant" values within the structure created by genetic search [384]. The group of Bir Bhanu has used GP to preprocess images, particularly of human faces,

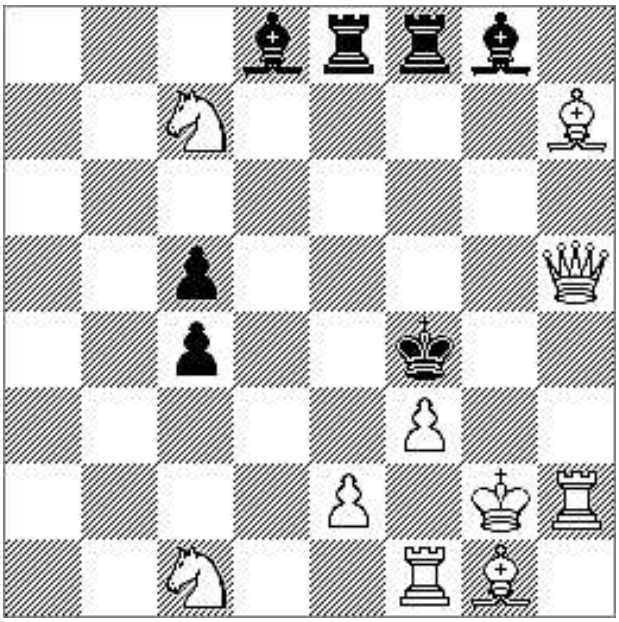

Figure 17: Example mate-in-2 problem.

to find regions of interest, for subsequent analysis [445]. (See also [412].) A particular strength of GP is its ability to take data from disparate sources [47, 399].

In New Zealand, Mengjie Zhang has been particularly active at evolving programs with GP to visually classify objects (typically coins) [453]. He has also applied GP to human speech [441]. "Parisian GP" is a system in which the image processing task is split across a swarm of evolving agents ("flies"). In [259, 260] the flies reconstruct three-dimensions from pairs of stereo images. In [259] 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 minimise 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 [408].

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 [65] such as Arabic [196] and oriental languages, handwriting [221, 79, 407, 117] (such as the MNIST examples) and other texts [356] and musical scores [346].

The scope for applications of GP to image and signal processing is almost unbounded. A promising area is medical imaging [318]. GP image techniques can also be used with sonar signals [269]. Offline work on images, includes security and verification. For example,

the group of Asifullah Khan has used GP to detect image watermarks which have been tampered with [417]. Whilst recent work by Yang Zhang [454] has incorporated multi-objective fitness into GP image processing.

In 1999 Riccardo Poli, Stefano 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 Modelling

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.

Shu-Heng Chen has written more than 60 papers on using GP in finance and economics. Recent papers include modelling of agents in stock markets [58], game theory [57], evolving trading rules for the S&P 500 [448] and forecasting the Heng-Sheng index [59].

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 computerised agents. GP in increasingly being used as part of these simulations of social systems.

Christopher Neely and Paul Weller of the US Federal Reserve Bank used GP to study intraday technical trading of foreign exchange to suggest the market is "efficient" and found no evidence of excess returns [293, 290, 292, 289]. This negative result was criticised by Tarbert and her co-workers [268]. Later work by Neely *et al.* [294] 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, Mak Kaboudan shows GP can forecast international currency exchange rates [175], stocks [174] and stock returns [173]. Edward Tsang and his co-workers continue to apply GP to a variety of financial arenas, including: betting [414], forecasting stock prices [119], studying markets [167] and arbitrage [267]. The group of Michael Dempster and HSBC also use GP in foreign exchange trading [82, 83, 15]. Nelishia Pillay has used GP in social studies and teaching aids in education, e.g. [315]. As well as trees [202] other types of GP have been used in finance, e.g. [296].

Since 1995 the International Conference on Computing in Economics and Finance

(CEF) has been held every year. It regularly attracts GP papers, many of which are online. In 2007 Tony Brabazon and Michael 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 the group of Author Kordon has been very active [51, 278, 198]. In [197] he describes where industrial GP stands now and how it will progress. Another active collaboration is that between Miha Kovacic and Joze Balic, who have used GP in the computer numerical control of industrial milling and cutting machinery [199]. The partnership of Larry Deschaine and Frank Francone [111] is most famous for their use of Discipulus [108] for detecting bomb fragments and unexploded ordnance UXO [84]. Discipulus has been used as an aid in the development of control systems for rubbish incinerators [86].

One of the earliest users of GP in control was Mark Willis' Chemical Engineering group in Newcastle. E.g. they used GP to model flow in a plasticating extruder [430]. They also modelled extruding food [275] and control of chemical reactions in continuous stirred tank reactors [372]. Peter Marenbach investigated GP in the control of biotech reactors [266]. In [431] Willis surveyed GP applications, including to control. Other GP applications to plastic include [42]. Daniel Lewin has applied GP to the control of an integrated circuit fabrication plant [251, 75]. Roberto Domingos worked on simulations of nuclear reactors (PWRs to be exact) to devise better ways of preventing xenon oscillations [91]. 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, Peter Fleming's group in Sheffield used multiobjective GP [357, 151] to reduce the cost of running aircraft jet engines [14, 102]. [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 Simon Handley [139] and John Koza [210] used GP to make predictions about the behaviour and properties of biological systems, principally proteins. Howard Oakley, a practising medical doctor, used GP to model blood flow in toes [302] as part of his long term interests in frost bite.

In 2002 Wolfgang Banzhaf and James A. Foster organised 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 Elena Marchiori and Dave Corne organised EvoBio: the European conference on evolutionary computation, machine learning and data mining in bioinformatics. EvoBio is held every year along side 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 [403, 168, 99, 418, 98, 129, 128, 141, 274], single nuclear polymorphisms [28, 375, 349] and Affymetrix GeneChip microarray data [78, 234, 100, 149, 152, 156, 252, 253, 446].

Douglas Kell and his colleagues in Aberystwyth have had great success in applying GP widely in bioinformatics (see infrared spectra above and [169, 123, 127, 439, 379, 184, 181, 183, 182, 77, 2]). Another very active group is that of Jason Moore and his colleagues at Vanderbilt [287, 355, 288, 354].

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 [104, 47, 29, 21, 130, 141, 424, 419, 144, 381, 124, 220].

7.7 Mixing GP with Other Techniques

GP can be hybridised with other techniques. Hitoshi Iba [161], Nikolay Nikolaev [295], and Byoung-Tak Zhang [451] have incorporated information theoretic and minimum description length ideas into GP fitness functions to provide a degree of regularisation and so avoid over fitting (and bloat, see Section 9.3). As mentioned in Section 5.4.3 computer language grammars can be incorporated into GP. Indeed Man Leung Wong [435, 433, 437, 434] 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 Section 2.

7.8 GP to Create Searchers and Solvers – Hyper-heuristics

Hyper-heuristics could simply be defined as "heuristics to choose other heuristics" [44]. A heuristic is considered as a rule of thumb or an educated guess that reduces the search required to find a solution. The difference between metaheuristics 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 hyperheuristic. For example, GP has evolved competitive SAT solvers [113, 191, 20, 19], state-of-the-art or better than state-of-the-art

bin packing algorithms [45, 46, 342], particle swarm optimisers [335, 325], evolutionary algorithms [303], and travelling-salesman-problem solvers [304, 185, 186, 187].

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 Karl Sims [383] and Bill Latham. Christian Jacob's work [164, 165] provide many examples. Since 2003 EvoMUSART has been held every year with EuroGP. [273] considers the recent state of play in evolutionary art and music. Many recent techniques will be described in [265].

Evolutionary music [410] has been dominated by Jazz [388] which is not to everyone's taste. An exception is Bach [103]. Most approaches to evolving music have made at least some use of interactive evolution [401] in which the fitness of programs is provided by users, often via the Internet [54, 5]. The limitation is almost always finding enough people willing to participate [229]. Funes reports experiments which attracted thousands of people via the Internet who were entertained by evolved Tron players [115]. Costelloe tried to reduce the human burden in [69]. Algorithmic approaches are also possible [64, 162].

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 Craig Reynold's swarming "boids" [350] 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, etc.). In 1997 Craig was awarded an Oscar.

7.10 Entertainment and Computer Games

Today the major usage of computers is interactive games [344]. There has been a little work on incorporating artificial intelligence into main stream 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, 420]. Since 2004 the annual CEC conference has included sessions on EC in games. After chairing the IEEE Symposium on Computational Intelligence and Games 2005, Essex University Simon Lucas founded the IEEE computational intelligence society's technical committee on games. GP features heavily in the Games TC's activities, e.g. Othello, Poker, Black Gammon, 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 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 [203, 26, 239] or some of the other

books in the 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, 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 behaviour 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 customising them for specific purposes (e.g., adding new, application specific genetic operators or implementing unusual, knowledge-based initialisation 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 from Bill Punch, ECJ from Sean Luke and DGPC from David Andre.

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 (e.g., 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 [122]. For this reason (and possibly others), one should use one of the anti-bloat mechanisms

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

described in Section 9.3.

There are methods to automatically simplify expressions (e.g., in Mathematica and Emacs). However, since in general there is an exponentially 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 [242]. 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, e.g. 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 chose 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. (E.g. 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 [242]. 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 [406].

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 [125] calls this the "take over time". This can be made quite formal [34, 93]. However, for tournament selection, a simple rule of thumb is often sufficient. If T is the tournament size, about $\log_T(\text{Pop size})$ generations are needed for the whole population to become descents of a single individual. E.g. if we use binary tournaments ($T = 2$), then "take over" will require about 10 generation for a population of 1024. Alternatively if we have a population of a 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.

Chris Gathercole investigated a number of ways of changing which training examples to use as the GP progressed [120, 121]. (Eric Siegel proposed a rather different implementation in [382].) 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 specialised individual which takes over the population at the expense of solutions which are viable on other parts of the training data. Dynamic Sub 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 [73] 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 [39, 436, 242, 432] 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. Sid Maxwell proposed [272] 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. E.g. 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.

Astro 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 weather it was ready or not. Astro

called this an "any time" approach [404]. 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 [389, 246].

A simple technique to speed up the evaluation of complex fitness functions is to organise 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 [330, 331, 323, 332] 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 [150, 16, 376, 367, 43, 38, 53, 432, 118, 90]. However, co-evolution 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 Hares populations being a famous example. There are various "hall of fame" techniques [116], 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 [138]. GP populations have enormous amounts

of common code [242, 244, 246]. 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 organised 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 this is true irrespective of whether we need to run the same subtree inside a different individual or at a different time (i.e. 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 (i.e., 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 crossover 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 [239, 334, 89, 63] 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 [276] is perhaps the best known GP system which has implemented fitness caches. As well as the original DAG implementation [138] other work has included [176, 63, 442].

In [242] 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, etc. 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 (e.g. 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" [8]. For example, when Ian Turton [306] ran GP on a Cray super computer 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.

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 super computer 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, etc. It was this geographic spread that led Sewell Wright [440] to proposed that geography and changes to it are of great importance to the formation of new species and so to natural evolution as a whole.

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 geographies bring 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 realise *physically-distributed* populations, one can obtain similar benefits by using *logically-distributed* populations in a single machine.

8.6.1 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 parallelisation 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.

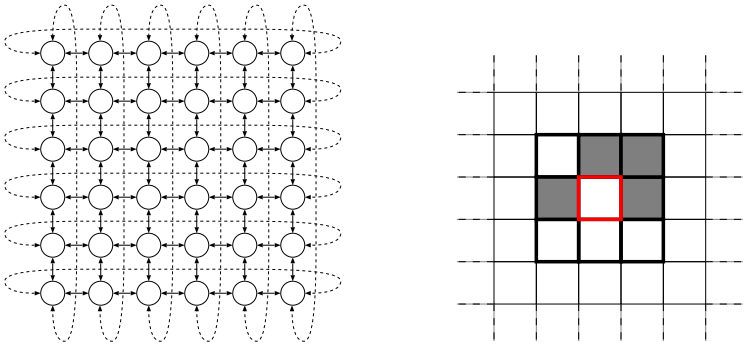

Figure 18: Spatially structured GP populations. (a) Toroidal grid of demes where each deme (a node) contains a subpopulation and demes periodically exchange a small group of high-fitness individuals using a grid of communication channels. (b) Fine-grained distributed GP, where each grid cell contains 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 (e.g., the individuals shaded) in its 3×3 neighbourhood.

In the master-slave model [311] breeding, selection crossover, mutation etc. 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 bottle neck. 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.

8.6.2 Geographically Distributed GP

As we have seen, unless some type of synchronisation or check pointing is imposed, e.g. 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 subpopulations (known as *demes* [67, 88, 242]) 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 parallelisation can bring benefits similar to those found in nature by Sewell Wright. 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 [7], David 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 Figure 18a)

Figure 19: A global population [231]. Straight lines show connections between major sites in a continuously evolving L-System.

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 [394]. Mitch Potter [343] argues in favour of spatial separation in populations (see Figure 18b). Dave Goldberg [126] also suggests low migration rates. In [427] Darrell Whitley includes some guidance on parallel GAs.

While many have glanced enviously at Koza's 1000 node Beowulf [397], a super computer [172, 31] 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 GP systems can be implemented using MPI [422] or PVM [105], the use of such tools is not necessary: simple Unix commands and port-to-port HTTP is sufficient [340]. 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 [231], ala seti@home, is perfectly feasible [62]. (See Figure 19). Other parallel GPs include [364, 362, 7, 107, 402, 270, 106, 70, 255, 135, 48, 55].

8.6.3 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 [312].

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 [286] 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 [261, 279, 95, 348, 140, 60, 233, 236]. So far, in GP, GPUs have just been used for fitness evaluation. Simon 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 [142] which was loaded onto his Laptop's GPU in order to run fitness cases. We used [233, 236] a SIMD interpreter [172] 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 linearised. We used used reverse polish notation (RPN). I.e. post fix notation, rather than pre-fix notation. RPN avoids recursive calls in the interpreter [233]. 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. Man-Leung Wong showed, for a genetic algorithm, these too can be done by the GPU [438].

Although each of the GPU's processors may be individually quite fast and the manufacturers claim huge aggregate FLOP ratings, the GPUs are optimised for graphics work. In practice it is hard to keep all the processors fully loaded. Nevertheless 30 GFLOP s^{-1} has been achieved [236]. 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 [236] and 12.6 [233].

8.7 GP Trouble-shooting

A number of practical recommendations for GP work can be made. To a large extent the advice in [195] and [203] 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. Recognising 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 (i.e., diversity of behaviour) 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. 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.[3]
  - 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.

[3]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.

- 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, e.g., 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 paper 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 behaviour varies from run to run. It is also complex adaptive system which sometimes shows complex and unexpected behaviours (such as bloat). So, it 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 behaviours, 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 *visualise* 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 visualise 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 behaviour 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, etc.). Then, one can try to suggest explanations about the behaviour 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 behaviour 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 [154, 429]. Schema theories are based on the idea of partitioning the search space into subsets, called *schemata*. They are concerned with modelling and explaining the dynamics of the distribution of the population over the schemata. Modern GA schema theory [395, 396] 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.[4]

The theory of schemata in GP has had a difficult childhood. Some excellent early efforts led to different worst-case-scenario schema theorems [203, 3, 309, 425, 326, 358]. Only very recently have the first exact schema theories become available [316, 320, 321] which give exact formulations (rather than lower bounds) for the expected number of individuals sampling a schema at the next generation. Initially [320, 321], these exact theories were only applicable to GP with one-point crossover (see Section 2.4). However, more recently they have been extended to the class of homologous crossovers [338] and to virtually all types of crossovers that swap subtrees [336, 337], 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 Section 2.4, as well as more constrained forms of crossover such as strongly-typed GP crossover (see Section 5.4.2), 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. I.e., schema theories do not tell us how fitness is distributed in the search space.

The characterisation of the space of computer programs explored by GP has been another main topic of theoretical research [239].[5] In this category are theoretical results

[4] Other models of evolutionary algorithms exist, such those based on Markov chain theory (e.g. [297, 76]) or on statistical mechanics (e.g. [345]). Only Markov models [341, 338, 284] have been applied to GP, but they are not as developed as schema theory.

[5] 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

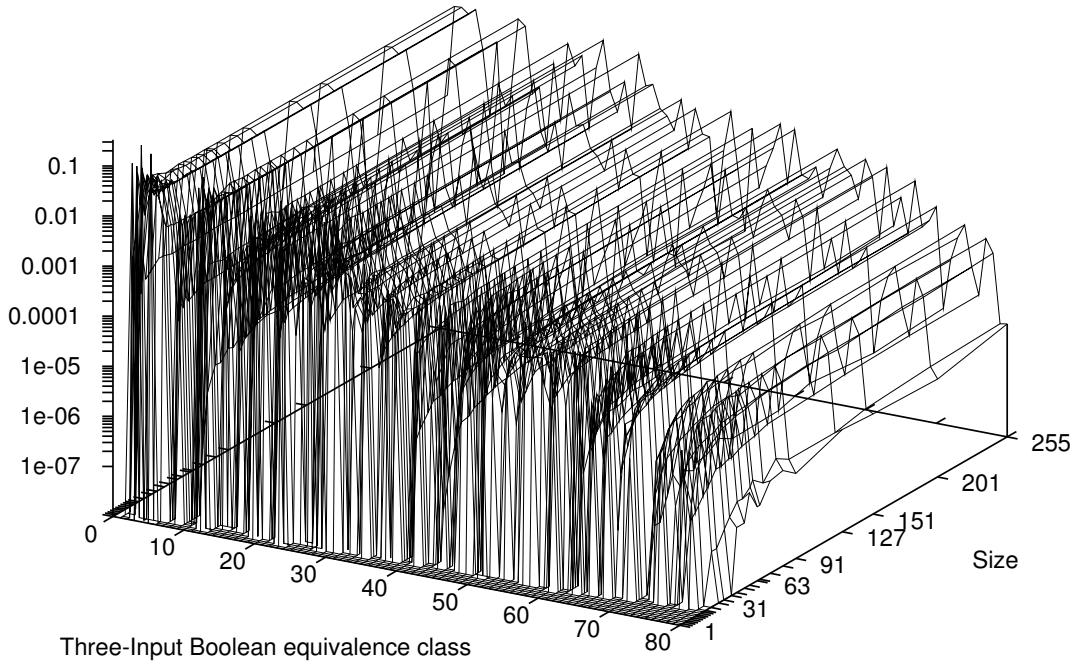

Figure 20: Proportion of NAND trees that yield each three-input functions. As circuit size increases the distribution approaches a limit.

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 Figure 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. 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 [239, 225, 226, 228, 227, 230]. 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).

software engineering to theoretical computer science.

Recently [333], we started extending our results to Turing complete machine code programs. 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 90s 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 generalisation 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, etc. 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 [277]: 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 [386]: *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 [237, 248]: 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 [334, 89]: 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 [248, 385]. E.g. size fair crossover or size fair mutation [243, 71], Tarpeian bloat control [322], parsimony pressure [450, 451, 452], 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 [307, 11]. There are also several mutation operators that may help control the average tree size in the population while still introducing new genetic material. [192] proposes a mutation operator which prevents the offspring's depth being more then 15% larger than its parent. [222] proposes two mutation operators in which the new random subtree is on average the same size as the code it replaces. In Hoist mutation [194] 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" [415] 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."

Turing did not specify how to conduct the "genetical or evolutionary search" for machine intelligence. In particular, he did not mention the idea of a population-based parallel search in conjunction with sexual recombination (crossover) as described in John Holland's 1975 book *Adaptation in Natural and Artificial Systems* [155]. However, in his 1950 paper "Computing Machinery and Intelligence" [416], 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"

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 (i.e., selective pressure in the form of what we now call "fitness").

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

Turing also understood the need to evaluate objectively the behaviour 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 summarised by Arthur Samuel's position statement quoted in the introduction of this paper. The eight criteria for human competitiveness we discussed in Section 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 Section 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

We would like to thank Rick Riolo for timely assistance.

Appendix: Resources

Following the publication of [203], 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 International Conference on Genetic Algorithms (ICGA-93, ICGA-95), the IEEE conferences 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 [291] to betting [414], from bomb detection [112] to architecture [308], from the steel industry to the environment [166], from space [258] to biology [168], and many others (as we have seen in Section 7). In 1996 it was possible to list (almost all) GP applications [241], but today the range is far too great and here we simply list some GP resources, which, we hope, will guide readers towards their goals.

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 "Genetic Programming" 1992 (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 book series edited by John Koza and Dave Goldberg now contains 14 books starting with "Genetic Programming and Data Structures" [242]. Apart from Jaws, these tended to be for the GP specialist. However, 1997 saw the introduction of the first text book dedicated to GP [26]. Eiben [96] and Goldberg [125] provide general treatment on evolutionary algorithms.

Other titles include: "Principia Evolvica – Simulierte Evolution mit Mathematica" (in German) [163] (English version [165]), "Data Mining Using Grammar Based Genetic Programming and Applications" [437], "Genetic Programming" (in Japanese) [160] and "Humanoider: Sjavlarande robotar och artificiell intelligens" (in Swedish) [301].

Readers interested in mathematical and empirical analyses of GP behaviour may find "Foundations of Genetic Programming" [239] useful.

Videos

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

Journals

In addition to GP's own "Genetic Programming and Evolvable Machines" journal, "Evolutionary Computation", the "IEEE transaction on Evolutionary Computation", "Complex Systems" and many others publish GP articles. The GP bibliography lists a further 375 different journals worldwide that have published articles related to GP.

Conference Proceedings

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 John Koza in 1996 in Stanford. Since 1999 it has been combined with the International Conference on Genetic Algorithms to form GECCO. 98% of GECCO papers are online. The Michigan based "Genetic Programming Theory and Practice" workshop [353, 310, 449, 351] will shortly publish its fifth proceedings [352]. Other EC conferences, such as CEC, PPSN, Evolution Artificielle and WSC, also regularly contain GP papers.

Examples

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, e.g., 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, admires 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 [108].

Online Resources

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 Figure 21). 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 organisation 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" and LISP code for implementing GP, as in *Genetic Programming* [203].

References

[1] Sameer H. Al-Sakran, John R. Koza, and Lee W. Jones. Automated re-invention of a previously patented optical lens system using genetic programming. In Maarten Keijzer, Andrea Tettamanzi, Pierre Collet, Jano I. van Hemert, and Marco Tomassini, editors, *Proceedings of the 8th European Conference on Genetic Programming*, volume 3447 of *Lecture Notes in Computer Science*, pages 25–37, Lausanne, Switzerland, 30 March - 1 April 2005. Springer.

[2] Jess Allen, Hazel M. Davey, David Broadhurst, Jim K. Heald, Jem J. Rowland, Stephen G. Oliver, and Douglas B. Kell. High-throughput classification of yeast mutants for functional genomics using metabolic footprinting. *Nature Biotechnology*, 21(6):692–696, June 2003.

[3] Lee Altenberg. Emergent phenomena in genetic programming. In Anthony V. Sebald and Lawrence J. Fogel, editors, *Evolutionary Programming — Proceedings of the Third Annual Conference*, pages 233–241, San Diego, CA, USA, 24-26 February 1994. World Scientific Publishing.

[4] Alexandre P. Alves da Silva and Pedro Jose Abrao. Applications of evolutionary computation in electric power systems. In David B. Fogel, Mohamed A. El-Sharkawi, Xin Yao, Garry Greenwood, Hitoshi Iba, Paul Marrow, and Mark Shackleton, editors, *Proceedings of the 2002 Congress on Evolutionary Computation CEC2002*, pages 1057–1062. IEEE Press, 2002.

[5] Daichi Ando, Palle Dahlsted, Mats Nordahl, and Hitoshi Iba. Interactive GP with tree representation of classical music pieces. In Mario Giacobini, Anthony

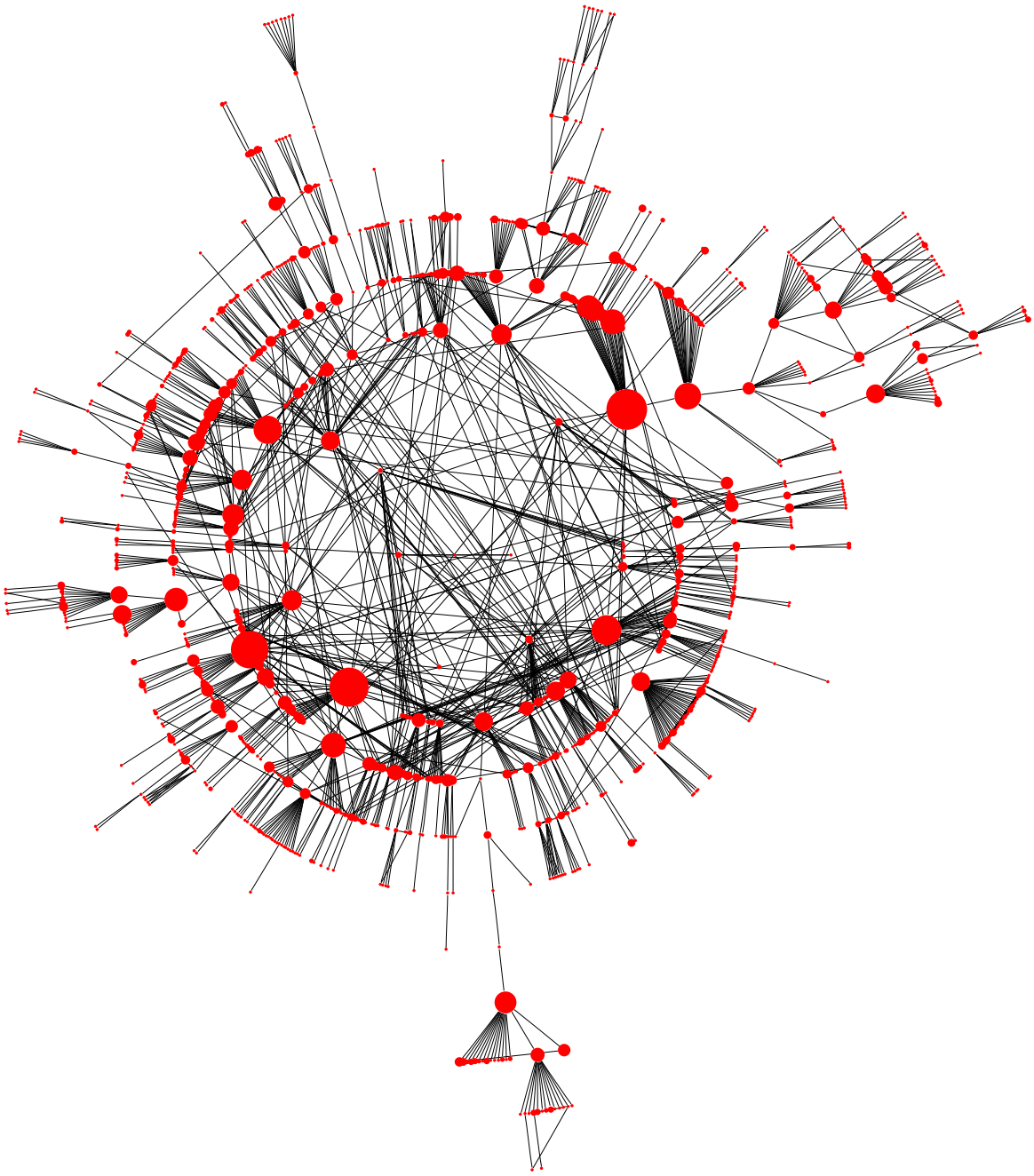

Figure 21: Co-author connections within GP. Each of the 1141 red dots indicates an author. The black lines links people who have co-authored one or more papers. (To reduce clutter only links to first author are shown.) The online version is annotated by JavaScript and contains hyperlinks to authors and 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" [411] 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 on line via the gp-bibliography www pages.

Brabazon, Stefano Cagnoni, Gianni A. Di Caro, Rolf Drechsler, Muddassar Farooq, Andreas Fink, Evelyne Lutton, Penousal Machado, Stefan Minner, Michael O'Neill, Juan Romero, Franz Rothlauf, Giovanni Squillero, Hideyuki Takagi, A. Sima Uyar, and Shengxiang Yang, editors, *Applications of Evolutionary Computing, EvoWorkshops2007: EvoCOMNET, EvoFIN, EvoIASP, EvoInteraction, EvoMUSART, EvoSTOC, EvoTransLog*, volume 4448 of *LNCS*, pages 577–584, Valencia, Spain, 11-13 April 2007. Springer Verlag.

[6] David Andre, Forrest H Bennett III, and John R. Koza. Discovery by genetic programming of a cellular automata rule that is better than any known rule for the majority classification problem. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 3–11, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[7] David Andre and John R. Koza. Parallel genetic programming: A scalable implementation using the transputer network architecture. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 16, pages 317–338. MIT Press, Cambridge, MA, USA, 1996.

[8] David Andre and John R. Koza. A parallel implementation of genetic programming that achieves super-linear performance. *Information Sciences*, 106(3-4):201–218, 1998.

[9] Peter J. Angeline. An investigation into the sensitivity of genetic programming to the frequency of leaf selection during subtree crossover. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 21–29, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[10] Peter J. Angeline. Subtree crossover: Building block engine or macromutation? In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 9–17, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[11] Peter J. Angeline. Multiple interacting programs: A representation for evolving complex behaviors. *Cybernetics and Systems*, 29(8):779–806, November 1998.

[12] Peter J. Angeline and K. E. Kinnear, Jr., editors. *Advances in Genetic Programming 2*. MIT Press, Cambridge, MA, USA, 1996.

[13] Peter J. Angeline and Jordan B. Pollack. The evolutionary induction of subroutines. In *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society*, pages 236–241, Bloomington, Indiana, USA, 1992. Lawrence Erlbaum.

[14] V. Arkov, C. Evans, P. J. Fleming, D. C. Hill, J. P. Norton, I. Pratt, D. Rees, and K. Rodriguez-Vazquez. System identification strategies applied to aircraft gas turbine engines. *Annual Reviews in Control*, 24(1):67–81, 2000.

[15] Mark P. Austin, Graham Bates, Michael A. H. Dempster, Vasco Leemans, and Stacy N. Williams. Adaptive systems for foreign exchange trading. *Quantitative Finance*, 4(4):37–45, August 2004.

[16] Yaniv Azaria and Moshe Sipper. GP-gammon: Genetically programming backgammon players. *Genetic Programming and Evolvable Machines*, 6(3):283–300, September 2005. Published online: 12 August 2005.

[17] Yaniv Azaria and Moshe Sipper. Using GP-gammon: Using genetic programming to evolve backgammon players. In Maarten Keijzer, Andrea Tettamanzi, Pierre Collet, Jano I. van Hemert, and Marco Tomassini, editors, *Proceedings of the 8th European Conference on Genetic Programming*, volume 3447 of *Lecture Notes in Computer Science*, pages 132–142, Lausanne, Switzerland, 30 March - 1 April 2005. Springer.

[18] Vladan Babovic. *Emergence, evolution, intelligence; Hydroinformatics - A study of distributed and decentralised computing using intelligent agents.* A. A. Balkema Publishers, Rotterdam, Holland, 1996.

[19] Mohamed Bader-El-Den and Riccardo Poli. Generating sat local-search heuristics using a gp hyper-heuristic framework. In *Proceedings of Evolution Artificielle*, October 2007.

[20] Mohamed Bahy Bader-El-Den and Riccardo Poli. A GP-based hyper-heuristic framework for evolving 3-SAT heuristics. In Dirk Thierens, Hans-Georg Beyer, Josh Bongard, Jurgen Branke, John Andrew Clark, Dave Cliff, Clare Bates Congdon, Kalyanmoy Deb, Benjamin Doerr, Tim Kovacs, Sanjeev Kumar, Julian F. Miller, Jason Moore, Frank Neumann, Martin Pelikan, Riccardo Poli, Kumara Sastry, Kenneth Owen Stanley, Thomas Stutzle, Richard A Watson, and Ingo Wegener, editors, *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 2, pages 1749–1749, London, 7-11 July 2007. ACM Press.

[21] William Bains, Richard Gilbert, Lilya Sviridenko, Jose-Miguel Gascon, Robert Scoffin, Kris Birchall, Inman Harvey, and John Caldwell. Evolutionary computational methods to predict oral bioavailability QSPRs. *Current Opinion in Drug Discovery and Development*, 5(1):44–51, January 2002.

[22] James E. Baker. Reducing bias and inefficiency in the selection algorithm. In John J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms and their Application*, pages 14–21, Cambridge, MA, USA, 1987. Lawrence Erlbaum Associates.

[23] Joze Balic. *Flexible Manufacturing Systems; Development - Structure - Operation - Handling - Tooling*. Manufacturing technology. DAAAM International, Vienna, 1999.

[24] Wolfgang Banzhaf. Genetic programming for pedestrians. In Stephanie Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, page 628, University of Illinois at Urbana-Champaign, 17-21 July 1993. Morgan Kaufmann.

[25] Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors. *GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference*, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[26] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, San Francisco, CA, USA, January 1998.

[27] Wolfgang Banzhaf, Riccardo Poli, Marc Schoenauer, and Terence C. Fogarty, editors. *Genetic Programming*, volume 1391 of *LNCS*, Paris, 14-15 April 1998. Springer-Verlag.

[28] S. J. Barrett. Recurring analytical problems within drug discovery and development. In Tobias Scheffer and Ulf Leser, editors, *Data Mining and Text Mining for Bioinformatics: Proceedings of the European Workshop*, pages 6–7, Dubrovnik, Croatia, 22 September 2003. Invited talk.

[29] S. J. Barrett and W. B. Langdon. Advances in the application of machine learning techniques in drug discovery, design and development. In Ashutosh Tiwari, Joshua Knowles, Erel Avineri, Keshav Dahal, and Rajkumar Roy, editors, *Applications of Soft Computing: Recent Trends*, Advances in Soft Computing, pages 99–110, On the World Wide Web, 19 September - 7 October 2005 2006. Springer.

[30] Forrest H Bennett III. Automatic creation of an efficient multi-agent architecture using genetic programming with architecture-altering operations. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 30–38, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[31] Forrest H Bennett III, John R. Koza, James Shipman, and Oscar Stiffelman. Building a parallel computer system for $18,000 that performs a half peta-flop per day. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1484–1490, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[32] Hans-Georg Beyer, Una-May O'Reilly, Dirk V. Arnold, Wolfgang Banzhaf, Christian Blum, Eric W. Bonabeau, Erick Cantu-Paz, Dipankar Dasgupta, Kalyanmoy Deb, James A. Foster, Edwin D. de Jong, Hod Lipson, Xavier Llora, Spiros Mancoridis, Martin Pelikan, Guenther R. Raidl, Terence Soule, Andy M. Tyrrell, Jean-Paul Watson, and Eckart Zitzler, editors. *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, Washington DC, USA, 25-29 June 2005. ACM Press.

[33] Bir Bhanu, Yingqiang Lin, and Krzysztof Krawiec. *Evolutionary Synthesis of Pattern Recognition Systems*. Monographs in Computer Science. Springer-Verlag, New York, 2005.

[34] Tobias Blickle. *Theory of Evolutionary Algorithms and Application to System Synthesis*. PhD thesis, Swiss Federal Institute of Technology, Zurich, November 1996.

[35] Anthony Brabazon and Michael O'Neill. *Biologically Inspired Algorithms for Financial Modelling*. Natural Computing Series. Springer, 2006.

[36] Markus Brameier and Wolfgang Banzhaf. A comparison of linear genetic programming and neural networks in medical data mining. *IEEE Transactions on Evolutionary Computation*, 5(1):17–26, February 2001.

[37] Markus Brameier and Wolfgang Banzhaf. *Linear Genetic Programming*. Number XVI in Genetic and Evolutionary Computation. Springer, 2007.

[38] Markus Brameier, Josien Haan, Andrea Krings, and Robert M MacCallum. Automatic discovery of cross-family sequence features associated with protein function. *BMC bioinformatics [electronic resource]*, 7(16), January 12 2006.

[39] Scott Brave. Evolving recursive programs for tree search. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 10, pages 203–220. MIT Press, Cambridge, MA, USA, 1996.

[40] Scott Brave and Annie S. Wu, editors. *Late Breaking Papers at the 1999 Genetic and Evolutionary Computation Conference*, Orlando, Florida, USA, 13 July 1999.

[41] Miran Brezocnik. *Uporaba genetskega programiranja v inteligentnih proizvodnih sistemih*. University of Maribor, Faculty of mechanical engineering, Maribor, Slovenia, 2000.

[42] Miran Brezocnik, Joze Balic, and Leo Gusel. Artificial intelligence approach to determination of flow curve. *Journal for technology of plasticity*, 25(1-2):1–7, 2000.

[43] Gunnar Buason, Nicklas Bergfeldt, and Tom Ziemke. Brains, bodies, and beyond: Competitive co-evolution of robot controllers, morphologies and environments. *Genetic Programming and Evolvable Machines*, 6(1):25–51, March 2005.

[44] E. Burke, G. Kendall, J. Newall, E. Hart, P. Ross, and S. Schulenburg. Hyper-heuristics: an emerging direction in modern search technology. In F. Glover and G. Kochenberger, editors, *Handbook of Metaheuristics*, pages 457–474. Kluwer Academic Publishers, 2003.

[45] E. K. Burke, M. R. Hyde, and G. Kendall. Evolving bin packing heuristics with genetic programming. In Thomas Philip Runarsson, Hans-Georg Beyer, Edmund Burke, Juan J. Merelo-Guervos, L. Darrell Whitley, and Xin Yao, editors, *Parallel Problem Solving from Nature - PPSN IX*, volume 4193 of *LNCS*, pages 860–869, Reykjavik, Iceland, 9-13 September 2006. Springer-Verlag.

[46] Edmund K. Burke, Matthew R. Hyde, Graham Kendall, and John Woodward. Automatic heuristic generation with genetic programming: evolving a jack-of-all-trades or a master of one. In Dirk Thierens, Hans-Georg Beyer, Josh Bongard, Jurgen Branke, John Andrew Clark, Dave Cliff, Clare Bates Congdon, Kalyanmoy Deb, Benjamin Doerr, Tim Kovacs, Sanjeev Kumar, Julian F. Miller, Jason Moore, Frank Neumann, Martin Pelikan, Riccardo Poli, Kumara Sastry, Kenneth Owen Stanley, Thomas Stutzle, Richard A Watson, and Ingo Wegener, editors, *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 2, pages 1559–1565, London, 7-11 July 2007. ACM Press.

[47] B. F. Buxton, W. B. Langdon, and S. J. Barrett. Data fusion by intelligent classifier combination. *Measurement and Control*, 34(8):229–234, October 2001.

[48] Stefano Cagnoni, Federico Bergenti, Monica Mordonini, and Giovanni Adorni. Evolving binary classifiers through parallel computation of multiple fitness cases. *IEEE Transactions on Systems, Man and Cybernetics - Part B*, 35(3):548–555, June 2005.

[49] Weihua Cai, Arturo Pacheco-Vega, Mihir Sen, and K. T. Yang. Heat transfer correlations by symbolic regression. *International Journal of Heat and Mass Transfer*, 49(23-24):4352–4359, November 2006.

[50] Erick Cantú-Paz, James A. Foster, Kalyanmoy Deb, Lawrence Davis, Rajkumar Roy, Una-May O'Reilly, Hans-Georg Beyer, Russell K. Standish, Graham Kendall, Stewart W. Wilson, Mark Harman, Joachim Wegener, Dipankar Dasgupta, Mitchell A. Potter, Alan C. Schultz, Kathryn A. Dowsland, Natasha Jonoska, and Julian F. Miller, editors. *Genetic and Evolutionary Computation – GECCO 2003, Part I*, volume 2723 of *Lecture Notes in Computer Science*, Chicago, IL, USA, 12-16 July 2003. Springer.

[51] Flor Castillo, Arthur Kordon, and Guido Smits. Robust pareto front genetic programming parameter selection based on design of experiments and industrial data. In Rick L. Riolo, Terence Soule, and Bill Worzel, editors, *Genetic Programming Theory and Practice IV*, volume 5 of *Genetic and Evolutionary Computation*, chapter 2, pages –. Springer, Ann Arbor, 11-13 May 2006.

[52] Malik Chami and Denis Robilliard. Inversion of oceanic constituents in case I and II waters with genetic programming algorithms. *Applied Optics*, 41(30):6260–6275, October 2002.

[53] Alastair Channon. Unbounded evolutionary dynamics in a system of agents that actively process and transform their environment. *Genetic Programming and Evolvable Machines*, 7(3):253–281, October 2006.

[54] Dennis L. Chao and Stephanie Forrest. Information immune systems. *Genetic Programming and Evolvable Machines*, 4(4):311–331, December 2003.

[55] Sin Man Cheang, Kwong Sak Leung, and Kin Hong Lee. Genetic parallel programming: Design and implementation. *Evolutionary Computation*, 14(2):129–156, Summer 2006.

[56] Shu-Heng Chen, editor. *Genetic Algorithms and Genetic Programming in Computational Finance*. Kluwer Academic Publishers, Dordrecht, July 2002.

[57] Shu-Heng Chen, John Duffy, and Chia-Hsuan Yeh. Equilibrium selection via adaptation: Using genetic programming to model learning in a coordination game. *The Electronic Journal of Evolutionary Modeling and Economic Dynamics*, 15 January 2002.

[58] Shu-Heng Chen and Chung-Chih Liao. Agent-based computational modeling of the stock price-volume relation. *Information Sciences*, 170(1):75–100, 18 February 2005.

[59] Shu-Heng Chen, Hung-Shuo Wang, and Byoung-Tak Zhang. Forecasting high-frequency financial time series with evolutionary neural trees: The case of hengsheng stock index. In Hamid R. Arabnia, editor, *Proceedings of the International Conference on Artificial Intelligence, IC-AI '99*, volume 2, pages 437–443, Las Vegas, Nevada, USA, 28 June-1 July 1999. CSREA Press.

[60] Darren M. Chitty. A data parallel approach to genetic programming using programmable graphics hardware. In Dirk Thierens, Hans-Georg Beyer, Josh Bongard, Jurgen Branke, John Andrew Clark, Dave Cliff, Clare Bates Congdon, Kalyanmoy Deb, Benjamin Doerr, Tim Kovacs, Sanjeev Kumar, Julian F. Miller, Jason Moore, Frank Neumann, Martin Pelikan, Riccardo Poli, Kumara Sastry, Kenneth Owen Stanley, Thomas Stutzle, Richard A Watson, and Ingo Wegener, editors, *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 2, pages 1566–1573, London, 7-11 July 2007. ACM Press.

[61] Sung-Bae Cho, Nguyen Xuan Hoai, and Yin Shan, editors. *Proceedings of The First Asian-Pacific Workshop on Genetic Programming*, Rydges (lakeside) Hotel, Canberra, Australia, 8 December 2003.

[62] Fuey Sian Chong and W. B. Langdon. Java based distributed genetic programming on the internet. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, page 1229, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann. Full text in technical report CSRP-99-7.

[63] Vic Ciesielski and Xiang Li. Analysis of genetic programming runs. In R I Mckay and Sung-Bae Cho, editors, *Proceedings of The Second Asian-Pacific Workshop on Genetic Programming*, Cairns, Australia, 6-7 December 2004.

[64] Rudi Cilibrasi, Paul Vitanyi, and Ronald de Wolf. Algorithmic clustering of music based on string compression. *Computer Music Journal*, 28(4):49–67, Winter 2004.

[65] Rudi Cilibrasi and Paul M. B. Vitanyi. Clustering by compression. *IEEE Transactions on Information Theory*, 51(4):1523–1545, April 2005.

[66] Pierre Collet, Marco Tomassini, Marc Ebner, Steven Gustafson, and Anikó Ekárt, editors. *Proceedings of the 9th European Conference on Genetic Programming*, volume 3905 of *Lecture Notes in Computer Science*, Budapest, Hungary, 10 - 12 April 2006. Springer.

[67] Robert J. Collins. *Studies in Artificial Evolution*. PhD thesis, UCLA, Artificial Life Laboratory, Department of Computer Science, University of California, Los Angeles, LA CA 90024, USA, 1992.

[68] F. Corno, E. Sanchez, and G. Squillero. Evolving assembly programs: how games help microprocessor validation. *Evolutionary Computation, IEEE Transactions on*, 9(6):695–706, 2005.

[69] Dan Costelloe and Conor Ryan. Towards models of user preferences in interactive musical evolution. In Dirk Thierens, Hans-Georg Beyer, Josh Bongard, Jurgen Branke, John Andrew Clark, Dave Cliff, Clare Bates Congdon, Kalyanmoy Deb, Benjamin Doerr, Tim Kovacs, Sanjeev Kumar, Julian F. Miller, Jason Moore, Frank Neumann, Martin Pelikan, Riccardo Poli, Kumara Sastry, Kenneth Owen Stanley, Thomas Stutzle, Richard A Watson, and Ingo Wegener, editors, *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 2, pages 2254–2254, London, 7-11 July 2007. ACM Press.

[70] Kyle Cranmer and R. Sean Bowman. PhysicsGP: A genetic programming approach to event selection. *Computer Physics Communications*, 167(3):165–176, 1 May 2005.

[71] Raphael Crawford-Marks and Lee Spector. Size control via size fair genetic operators in the PushGP genetic programming system. In W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and

N. Jonoska, editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 733–739, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[72] Ronald L. Crepeau. Genetic evolution of machine language software. In Justinian P. Rosca, editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 121–134, Tahoe City, California, USA, 9 July 1995.

[73] Robert Curry, Peter Lichodzijewski, and Malcolm I. Heywood. 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, August 2007.

[74] Jason M. Daida, Jonathan D. Hommes, Tommaso F. Bersano-Begey, Steven J. Ross, and John F. Vesecky. Algorithm discovery using the genetic programming paradigm: Extracting low-contrast curvilinear features from SAR images of arctic ice. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 21, pages 417–442. MIT Press, Cambridge, MA, USA, 1996.

[75] Eyal Dassau, Benyamin Grosman, and Daniel R. Lewin. Modeling and temperature control of rapid thermal processing. *Computers and Chemical Engineering*, 30(4):686–697, 15 February 2006.

[76] Thomas E. Davis and Jose C. Principe. A Markov chain framework for the simple genetic algorithm. *Evolutionary Computation*, 1(3):269–288, 1993.

[77] Jennifer P. Day, Douglas B. Kell, and Gareth W. Griffith. Differentiation of phytophthora infestans sporangia from other airborne biological particles by flow cytometry. *Applied and Environmental Microbiology*, 68(1):37–45, January 2002.

[78] Janaina S. de Sousa, Lalinka de C. T. Gomes, George B. Bezerra, Leandro N. de Castro, and Fernando J. Von Zuben. An immune-evolutionary algorithm for multiple rearrangements of gene expression data. *Genetic Programming and Evolvable Machines*, 5(2):157–179, June 2004.

[79] C. De Stefano, A. Della Cioppa, and A. Marcelli. Character preclassification based on genetic programming. *Pattern Recognition Letters*, 23(12):1439–1448, 2002.

[80] Kalyanmoy Deb. *Multi-objective optimization using evolutionary algorithms*. Wiley, 2001.

[81] Kalyanmoy Deb, Riccardo Poli, Wolfgang Banzhaf, Hans-Georg Beyer, Edmund Burke, Paul Darwen, Dipankar Dasgupta, Dario Floreano, James Foster, Mark Harman, Owen Holland, Pier Luca Lanzi, Lee Spector, Andrea Tettamanzi, Dirk Thierens, and Andy Tyrrell, editors. *Genetic and Evolutionary Computation – GECCO-2004, Part I*, volume 3102 of *Lecture Notes in Computer Science*, Seattle, WA, USA, 26-30 June 2004. Springer-Verlag.

[82] M. A. H. Dempster and C. M. Jones. A real-time adaptive trading system using genetic programming. *Quantitative Finance*, 1:397–413, 2000.

[83] M. A. H. Dempster, Tom W. Payne, Yazann Romahi, and G. W. P. Thompson. Computational learning techniques for intraday FX trading using popular technical indicators. *IEEE Transactions on Neural Networks*, 12(4):744–754, July 2001.

[84] Larry Deschaine. 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, May 2006.

[85] Larry M. Deschaine, Richard A. Hoover, Joseph N. Skibinski, Janardan J. Patel, Frank Francone, Peter Nordin, and M. J. Ades. 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, 14-18 April 2002.

[86] Larry M. Deschaine, Janardan J. Patel, Ronald D. Guthrie, Joseph T. Grimski, and M. J. Ades. Using linear genetic programming to develop a C/C++ simulation model of a waste incinerator. In M. Ades, editor, *Advanced Technology Simulation Conference*, Seattle, 22-26 April 2001.

[87] Patrik D'haeseleer. Context preserving crossover in genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 256–261, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.

[88] Patrik D'haeseleer and Jason Bluming. Effects of locality in individual and population evolution. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 8, pages 177–198. MIT Press, 1994.

[89] Stephen Dignum and Riccardo Poli. Generalisation of the limiting distribution of program sizes in tree-based genetic programming and analysis of its effects on bloat. In Dirk Thierens, Hans-Georg Beyer, Josh Bongard, Jurgen Branke, John Andrew Clark, Dave Cliff, Clare Bates Congdon, Kalyanmoy Deb, Benjamin Doerr, Tim Kovacs, Sanjeev Kumar, Julian F. Miller, Jason Moore, Frank Neumann, Martin Pelikan, Riccardo Poli, Kumara Sastry, Kenneth Owen Stanley, Thomas Stutzle, Richard A Watson, and Ingo Wegener, editors, *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 2, pages 1588–1595, London, 7-11 July 2007. ACM Press.

[90] J. U. Dolinsky, I. D. Jenkinson, and G. J. Colquhoun. Application of genetic programming to the calibration of industrial robots. *Computers in Industry*, 58(3):255–264, April 2007.

[91] Roberto P. Domingos, Roberto Schirru, and Aquilino Senra Martinez. Soft computing systems applied to PWR's xenon. *Progress in Nuclear Energy*, 46(3-4):297–308, 2005.

[92] Dimitris C. Dracopoulos. *Evolutionary Learning Algorithms for Neural Adaptive Control*. Perspectives in Neural Computing. Springer Verlag, P.O. Box 31 13 40, D-10643 Berlin, Germany, August 1997.

[93] Stefan Droste, Thomas Jansen, Günter Rudolph, Hans-Paul Schwefel, Karsten Tinnefeld, and Ingo Wegener. Theory of evolutionary algorithms and genetic programming. In Hans-Paul Schwefel, Ingo Wegener, and Klaus Weinert, editors, *Advances in Computational Intelligence: Theory and Practice*, Natural Computing Series, chapter 5, pages 107–144. Springer, 2003.

[94] Marc Ebner, Michael O'Neill, Anikó Ekárt, Leonardo Vanneschi, and Anna Isabel Esparcia-Alcázar, editors. *Proceedings of the 10th European Conference on Genetic Programming*, volume 4445 of *Lecture Notes in Computer Science*, Valencia, Spain, 11 - 13 April 2007. Springer.

[95] Marc Ebner, Markus Reinhardt, and Jürgen Albert. Evolution of vertex and pixel shaders. In Maarten Keijzer, Andrea Tettamanzi, Pierre Collet, Jano I. van Hemert, and Marco Tomassini, editors, *Proceedings of the 8th European Conference on Genetic Programming*, volume 3447 of *Lecture Notes in Computer Science*, pages 261–270, Lausanne, Switzerland, 30 March - 1 April 2005. Springer.

[96] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. Springer, 2003.

[97] Sven E. Eklund. A massively parallel GP engine in VLSI. In David B. Fogel, Mohamed A. El-Sharkawi, Xin Yao, Garry Greenwood, Hitoshi Iba, Paul Marrow, and Mark Shackleton, editors, *Proceedings of the 2002 Congress on Evolutionary Computation CEC2002*, pages 629–633. IEEE Press, 2002.

[98] David I. Ellis, David Broadhurst, and Royston Goodacre. 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, 2004.

[99] David I. Ellis, David Broadhurst, Douglas B. Kell, Jem J. Rowland, and Royston Goodacre. 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, June 2002.

[100] R. Eriksson and B. Olsson. Adapting genetic regulatory models by genetic programming. *Biosystems*, 76(1-3):217–227, 2004.

[101] Anna I. Esparcia-Alcazar and Kenneth C. Sharman. Genetic programming techniques that evolve recurrent neural networks architectures for signal processing. In *IEEE Workshop on Neural Networks for Signal Processing*, Seiko, Kyoto, Japan, September 1996.

[102] C. Evans, P. J. Fleming, D. C. Hill, J. P. Norton, I. Pratt, D. Rees, and K. Rodriguez-Vazquez. Application of system identification techniques to aircraft gas turbine engines. *Control Engineering Practice*, 9(2):135–148, 2001.

[103] Francine Federman, Gayle Sparkman, and Stephanie Watt. Representation of music in a learning classifier system utilizing bach chorales. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 1, page 785, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[104] Michael J. Felton. Survival of the fittest in drug design. *Modern Drug Discovery*, 3(9):49–50, November/December 2000.

[105] F. Fernandez, J. M. Sanchez, M. Tomassini, and J. A. Gomez. A parallel genetic programming tool based on PVM. In J. Dongarra, E. Luque, and T. Margalef, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proceedings of the 6th European PVM/MPI Users' Group Meeting*, volume 1697 of *Lecture Notes in Computer Science*, pages 241–248, Barcelona, Spain, September 1999. Springer-Verlag.

[106] Francisco Fernandez, Marco Tomassini, and Leonardo Vanneschi. An empirical study of multipopulation genetic programming. *Genetic Programming and Evolvable Machines*, 4(1):21–51, March 2003.

[107] Gianluigi Folino, Clara Pizzuti, and Giandomenico Spezzano. A scalable cellular implementation of parallel genetic programming. *IEEE Transactions on Evolutionary Computation*, 7(1):37–53, February 2003.

[108] James A. Foster. Review: Discipulus: A commercial genetic programming system. *Genetic Programming and Evolvable Machines*, 2(2):201–203, June 2001.

[109] James A. Foster, Evelyne Lutton, Julian Miller, Conor Ryan, and Andrea G. B. Tettamanzi, editors. *Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002*, volume 2278 of *LNCS*, Kinsale, Ireland, 3-5 April 2002. Springer-Verlag.

[110] Frank D. Francone, Markus Conrads, Wolfgang Banzhaf, and Peter Nordin. Homologous crossover in genetic programming. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1021–1026, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[111] Frank D. Francone and Larry M. Deschaine. 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, April 2004.

[112] Frank D. Francone, Larry M. Deschaine, and Jeffrey J. Warren. Discrimination of munitions and explosives of concern at F.E. warren AFB using linear genetic programming. In Dirk Thierens, Hans-Georg Beyer, Josh Bongard, Jurgen Branke, John Andrew Clark, Dave Cliff, Clare Bates Congdon, Kalyanmoy Deb, Benjamin Doerr, Tim Kovacs, Sanjeev Kumar, Julian F. Miller, Jason Moore, Frank Neumann, Martin Pelikan, Riccardo Poli, Kumara Sastry, Kenneth Owen Stanley, Thomas Stutzle, Richard A Watson, and Ingo Wegener, editors, *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 2, pages 1999–2006, London, 7-11 July 2007. ACM Press.

[113] Alex Fukunaga. Automated discovery of composite SAT variable selection heuristics. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 641–648, 2002.

[114] Alex S. Fukunaga. Evolving local search heuristics for SAT using genetic programming. In Kalyanmoy Deb, Riccardo Poli, Wolfgang Banzhaf, Hans-Georg Beyer, Edmund Burke, Paul Darwen, Dipankar Dasgupta, Dario Floreano, James Foster, Mark Harman, Owen Holland, Pier Luca Lanzi, Lee Spector, Andrea Tettamanzi, Dirk Thierens, and Andy Tyrrell, editors, *Genetic and Evolutionary Computation – GECCO-2004, Part II*, volume 3103 of *Lecture Notes in Computer Science*, pages 483–494, Seattle, WA, USA, 26-30 June 2004. Springer-Verlag.

[115] Pablo Funes, Elizabeth Sklar, Hugues Juille, and Jordan Pollack. Animal-animat coevolution: Using the animal population as fitness function. In Rolf Pfeifer, Bruce Blumberg, Jean-Arcady Meyer, and Stewart W. Wilson, editors, *From Animals to Animats 5: Proceedings of the Fifth International Conference on Simulation of Adaptive Behavior.*, pages 525–533, Zurich, Switzerland, August 17-21 1998. MIT Press.

[116] Pablo Funes, Elizabeth Sklar, Hugues Juille, and Jordan Pollack. Animal-animat coevolution: Using the animal population as fitness function. In Rolf Pfeifer, Bruce Blumberg, Jean-Arcady Meyer, and Stewart W. Wilson, editors, *From Animals to Animats 5: Proceedings of the Fifth International Conference on Simulation of Adaptive Behavior*, pages 525–533, Zurich, Switzerland, August 17-21 1998. MIT Press.

[117] Christian Gagne and Marc Parizeau. Genetic engineering of hierarchical fuzzy regional representations for handwritten character recognition. *International Journal on Document Analysis and Recognition*, 8(4):223–231, September 2006.

[118] Christian Gagné and Marc Parizeau. Co-evolution of nearest neighbor classifiers. *International Journal of Pattern Recognition and Artificial Intelligence*, 21(5):921–946, August 2007.

[119] Alma Lilia Garcia-Almanza and Edward P. K. Tsang. Forecasting stock prices using genetic programming and chance discovery. In *12th International Conference On Computing In Economics And Finance*, page number 489, July 2006.

[120] Chris Gathercole and Peter Ross. Dynamic training subset selection for supervised learning in genetic programming. In Yuval Davidor, Hans-Paul Schwefel, and Reinhard Männer, editors, *Parallel Problem Solving from Nature III*, volume 866 of *LNCS*, pages 312–321, Jerusalem, 9-14 October 1994. Springer-Verlag.

[121] Chris Gathercole and Peter Ross. Tackling the boolean even N parity problem with genetic programming and limited-error fitness. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 119–127, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[122] Sylvain Gelly, Olivier Teytaud, Nicolas Bredeche, and Marc Schoenauer. Universal consistency and bloat in GP. *Revue d'Intelligence Artificielle*, 20(6):805–827, 2006. Issue on New Methods in Machine Learning. Theory and Applications.

[123] Richard J. Gilbert, Royston Goodacre, Andrew M. Woodward, and Douglas B. Kell. Genetic programming: A novel method for the quantitative analysis of pyrolysis mass spectral data. *ANALYTICAL CHEMISTRY*, 69(21):4381–4389, 1997.

[124] Al Globus, John Lawton, and Todd Wipke. Automatic molecular design using evolutionary techniques. In Al Globus and Deepak Srivastava, editors, *The Sixth Foresight Conference on Molecular Nanotechnology*, Westin Hotel in Santa Clara, CA, USA, November 12-15, 1998 1998.

[125] David E. Goldberg. *Genetic Algorithms in Search Optimization and Machine Learning*. Addison-Wesley, 1989.

[126] David E. Goldberg, Hillol Kargupta, Jeffrey Horn, and Erick Cantu-Paz. Critical deme size for serial and parallel genetic algorithms. Technical report, Illinois Genetic Algorithms Laboratory, Department of General Engineering, University of Illinois at Urbana-Champaign, Il 61801, USA, 1995. IlliGAL Report no 95002.

[127] R. Goodacre and R. J. Gilbert. The detection of caffeine in a variety of beverages using curie-point pyrolysis mass spectrometry and genetic programming. *The Analyst*, 124:1069–1074, 1999.

[128] Royston Goodacre. Explanatory analysis of spectroscopic data using machine learning of simple, interpretable rules. *Vibrational Spectroscopy*, 32(1):33–45, 5 August 2003. 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.

[129] Royston Goodacre, Beverley Shann, Richard J. Gilbert, Eadaoin M. Timmins, Aoife C. McGovern, Bjorn K. Alsberg, Douglas B. Kell, and Niall A. Logan. 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, 1 January 2000.

[130] Royston Goodacre, Seetharaman Vaidyanathan, Warwick B. Dunn, George G. Harrigan, and Douglas B. Kell. Metabolomics by numbers: acquiring and understanding global metabolite data. *Trends in Biotechnology*, 22(5):245–252, 1 May 2004.

[131] F. Gruau. *Neural Network Synthesis using Cellular Encoding and the Genetic Algorithm.* PhD thesis, Laboratoire de l'Informatique du Parallilisme, Ecole Normale Supirieure de Lyon, France, 1994.

[132] F. Gruau and D. Whitley. Adding learning to the cellular development process: a comparative study. *Evolutionary Computation*, 1(3):213–233, 1993.

[133] Frederic Gruau. Genetic micro programming of neural networks. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 24, pages 495–518. MIT Press, 1994.

[134] Frederic Gruau. On using syntactic constraints with genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 19, pages 377–394. MIT Press, Cambridge, MA, USA, 1996.

[135] Steven Gustafson and Edmund K. Burke. The speciating island model: An alternative parallel evolutionary algorithm. *Journal of Parallel and Distributed Computing*, 66(8):1025–1036, August 2006. Parallel Bioinspired Algorithms.

[136] Steven Gustafson, Edmund K. Burke, and Natalio Krasnogor. On improving genetic programming for symbolic regression. In David Corne, Zbigniew Michalewicz, Marco Dorigo, Gusz Eiben, David Fogel, Carlos Fonseca, Garrison Greenwood, Tan Kay Chen, Guenther Raidl, Ali Zalzala, Simon Lucas, Ben Paechter, Jennifier Willies, Juan J. Merelo Guervos, Eugene Eberbach, Bob McKay, Alastair Channon, Ashutosh Tiwari, L. Gwenn Volkert, Dan Ashlock, and Marc Schoenauer, editors, *Proceedings of the 2005 IEEE Congress on Evolutionary Computation*, volume 1, pages 912–919, Edinburgh, UK, 2-5 September 2005. IEEE Press.

[137] R. J. Hampo and K. A. Marko. Application of genetic programming to control of vehicle systems. In *Proceedings of the Intelligent Vehicles '92 Symposium*, 1992. June 29 - July 1, 1992, Detroit, Mi, USA.

[138] S. Handley. 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*, volume 1, pages 154–159, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.

[139] Simon Handley. Automatic learning of a detector for alpha-helices in protein sequences via genetic programming. In Stephanie Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, pages 271–278, University of Illinois at Urbana-Champaign, 17-21 July 1993. Morgan Kaufmann.

[140] Simon Harding and Wolfgang Banzhaf. Fast genetic programming on GPUs. In Marc Ebner, Michael O'Neill, Anikó Ekárt, Leonardo Vanneschi, and Anna Isabel Esparcia-Alcázar, editors, *Proceedings of the 10th European Conference on Genetic Programming*, volume 4445 of *Lecture Notes in Computer Science*, pages 90–101, Valencia, Spain, 11 - 13 April 2007. Springer.

[141] George G. Harrigan, Roxanne H. LaPlante, Greg N. Cosma, Gary Cockerell, Royston Goodacre, Jane F. Maddox, James P. Luyendyk, Patricia E. Ganey, and Robert A. Roth. 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, 2 February 2004.

[142] Christopher Harris and Bernard Buxton. GP-COM: A distributed, component-based genetic programming system in C++. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, page 425, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[143] Brad Harvey, James Foster, and Deborah Frincke. Towards byte code genetic programming. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, page 1234, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[144] Samiul Hasan, Sabine Daugelat, P. S. Srinivasa Rao, and Mark Schreiber. Prioritizing genomic drug targets in pathogens: Application to mycobacterium tuberculosis. *PLoS Computational Biology*, 2(6):e61, June 2006.

[145] Ami Hauptman and Moshe Sipper. GP-endchess: Using genetic programming to evolve chess endgame players. In Maarten Keijzer, Andrea Tettamanzi, Pierre Collet, Jano I. van Hemert, and Marco Tomassini, editors, *Proceedings of the 8th European Conference on Genetic Programming*, volume 3447 of *Lecture Notes in Computer Science*, pages 120–131, Lausanne, Switzerland, 30 March - 1 April 2005. Springer.

[146] Ami Hauptman and Moshe Sipper. Evolution of an efficient search algorithm for the mate-in-N problem in chess. In Marc Ebner, Michael O'Neill, Anikó Ekárt, Leonardo Vanneschi, and Anna Isabel Esparcia-Alcázar, editors, *Proceedings of the 10th European Conference on Genetic Programming*, volume 4445 of *Lecture Notes in Computer Science*, pages 78–89, Valencia, Spain, 11 - 13 April 2007. Springer.

[147] Thomas Haynes, Roger Wainwright, Sandip Sen, and Dale Schoenefeld. Strongly typed genetic programming in evolving cooperation strategies. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 271–278, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.

[148] Thomas D. Haynes, Dale A. Schoenefeld, and Roger L. Wainwright. Type inheritance in strongly typed genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 18, pages 359–376. MIT Press, Cambridge, MA, USA, 1996.

[149] A Geert Heidema, Jolanda M A Boer, Nico Nagelkerke, Edwin C M Mariman, Daphne L van der A, and Edith J M Feskens. The challenge for genetic epidemiologists: how to analyze large numbers of SNPs in relation to complex diseases. *BMC Genetics*, 7(23), April 21 2006.

[150] W. Daniel Hillis. Co-evolving parasites improve simulated evolution as an optimization procedure. In Christopher G. Langton, Charles E. Taylor, J. Doyne Farmer, and Steen Rasmussen, editors, *Artificial Life II*, volume X of *Santa Fe Institute Studies in the Sciences of Complexity*, pages 313–324. Addison-Wesley, Santa Fe Institute, New Mexico, USA, February 1990 1992.

[151] Mark P. Hinchliffe and Mark J. Willis. Dynamic systems modelling using genetic programming. *Computers & Chemical Engineering*, 27(12):1841–1854, 2003.

[152] Shinn-Ying Ho, Chih-Hung Hsieh, Hung-Ming Chen, and Hui-Ling Huang. Interpretable gene expression classifier with an accurate and compact fuzzy rule base for microarray data analysis. *Biosystems*, 85(3):165–176, September 2006.

[153] Nguyen Xuan Hoai, R. I. McKay, and H. A. Abbass. Tree adjoining grammars, language bias, and genetic programming. In Conor Ryan, Terence Soule, Maarten Keijzer, Edward Tsang, Riccardo Poli, and Ernesto Costa, editors, *Genetic Programming, Proceedings of EuroGP'2003*, volume 2610 of *LNCS*, pages 335–344, Essex, 14-16 April 2003. Springer-Verlag.

[154] J. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, USA, 1975.

[155] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence.* MIT Press, 1992. First Published by University of Michigan Press 1975.

[156] Jin-Hyuk Hong and Sung-Bae Cho. The classification of cancer based on DNA microarray data that uses diverse ensemble genetic programming. *Artificial Intelligence In Medicine*, 36(1):43–58, January 2006.

[157] Daniel Howard and Simon C. Roberts. Incident detection on highways. In Una-May O'Reilly, Tina Yu, Rick L. Riolo, and Bill Worzel, editors, *Genetic Programming Theory and Practice II*, chapter 16, pages 263–282. Springer, Ann Arbor, 13-15 May 2004.

[158] Daniel Howard, Simon C. Roberts, and Richard Brankin. Target detection in imagery by genetic programming. *Advances in Engineering Software*, 30(5):303–311, 1999.

[159] Daniel Howard, Simon C. Roberts, and Conor Ryan. Pragmatic genetic programming strategy for the problem of vehicle detection in airborne reconnaissance. *Pattern Recognition Letters*, 27(11):1275–1288, August 2006. Evolutionary Computer Vision and Image Understanding.

[160] Hitoshi Iba. *Genetic Programming*. Tokyo Denki University Press, 1996.

[161] Hitoshi Iba, Hugo de Garis, and Taisuke Sato. Genetic programming using a minimum description length principle. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 12, pages 265–284. MIT Press, 1994.

[162] Yoshiyuki Inagaki. On synchronized evolution of the network of automata. *IEEE Transactions on Evolutionary Computation*, 6(2):147–158, April 2002.

[163] Christian Jacob. *Principia Evolvica – Simulierte Evolution mit Mathematica*. dpunkt.verlag, Heidelberg, Germany, August 1997.

[164] Christian Jacob. The art of genetic programming. *IEEE Intelligent Systems*, 15(3):83–84, May-June 2000.

[165] Christian Jacob. *Illustrating Evolutionary Computation with Mathematica*. Morgan Kaufmann, 2001.

[166] Kwang-Seuk Jeong, Dong-Kyun Kim, Peter Whigham, and Gea-Jae Joo. Modelling microcystis aeruginosa bloom dynamics in the nakdong river by means of evolutionary computation and statistical approach. *Ecological Modelling*, 161(1-2):67–78, 1 March 2003.

[167] Nanlin Jin and Edward Tsang. Co-adaptive strategies for sequential bargaining problems with discount factors and outside options. In *Proceedings of the 2006 IEEE Congress on Evolutionary Computation*, pages 7913–7920, Vancouver, 6-21 July 2006. IEEE Press.

[168] Helen E. Johnson, Richard J. Gilbert, Michael K. Winson, Royston Goodacre, Aileen R. Smith, Jem J. Rowland, Michael A. Hall, and Douglas B. Kell. Explanatory analysis of the metabolome using genetic programming of simple, interpretable rules. *Genetic Programming and Evolvable Machines*, 1(3):243–258, July 2000.

[169] Alun Jones, Daniella Young, Janet Taylor, Douglas B. Kell, and Jem J Rowland. Quantification of microbial productivity via multi-angle light scattering and supervised learning. *Biotechnology and Bioengineering*, 59(2):131–143, 20 July 1998.

[170] *Proceedings of the 2001 Congress on Evolutionary Computation CEC2001*, COEX, World Trade Center, 159 Samseong-dong, Gangnam-gu, Seoul, Korea, 27-30 May 2001. IEEE Press.

[171] Elsa Jordaan, Arthur Kordon, Leo Chiang, and Guido Smits. Robust inferential sensors based on ensemble of predictors generated by genetic programming. In Xin Yao, Edmund Burke, Jose A. Lozano, Jim Smith, Juan J. Merelo-Guervós, John A. Bullinaria, Jonathan Rowe, Peter Tiňo Ata Kabán, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature - PPSN VIII*, volume 3242 of *LNCS*, pages 522–531, Birmingham, UK, 18-22 September 2004. Springer-Verlag.

[172] Hugues Juille and Jordan B. Pollack. Massively parallel genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 17, pages 339–358. MIT Press, Cambridge, MA, USA, 1996.

[173] M. Kaboudan. A measure of time series predictability using genetic programming applied to stock returns. *Journal of Forecasting*, 18:345–357, 1999.

[174] M. A. Kaboudan. Genetic programming prediction of stock prices. *Computational Economics*, 6(3):207–236, December 2000.

[175] Mak Kaboudan. Extended daily exchange rates forecasts using wavelet temporal resolutions. *New Mathematics and Natural Computing*, 1:79–107, 2005.

[176] Maarten Keijzer. Efficiently representing populations in genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 13, pages 259–278. MIT Press, Cambridge, MA, USA, 1996.

[177] Maarten Keijzer. Scaled symbolic regression. *Genetic Programming and Evolvable Machines*, 5(3):259–269, September 2004.

[178] Maarten Keijzer, Mike Cattolico, Dirk Arnold, Vladan Babovic, Christian Blum, Peter Bosman, Martin V. Butz, Carlos Coello Coello, Dipankar Dasgupta, Sevan G. Ficici, James Foster, Arturo Hernandez-Aguirre, Greg Hornby, Hod Lipson, Phil McMinn, Jason Moore, Guenther Raidl, Franz Rothlauf, Conor Ryan, and Dirk Thierens, editors. *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, Seattle, Washington, USA, 8-12 July 2006. ACM Press.

[179] Maarten Keijzer, Una-May O'Reilly, Simon M. Lucas, Ernesto Costa, and Terence Soule, editors. *Genetic Programming 7th European Conference, EuroGP 2004, Proceedings*, volume 3003 of *LNCS*, Coimbra, Portugal, 5-7 April 2004. Springer-Verlag.

[180] Maarten Keijzer, Andrea Tettamanzi, Pierre Collet, Jano I. van Hemert, and Marco Tomassini, editors. *Proceedings of the 8th European Conference on Genetic Programming*, volume 3447 of *Lecture Notes in Computer Science*, Lausanne, Switzerland, 30 March - 1 April 2005. Springer.

[181] Douglas Kell. Defence against the flood. *Bioinformatics World*, pages 16–18, January/February 2002.

[182] Douglas B. Kell. Genotype-phenotype mapping: genes as computer programs. *Trends in Genetics*, 18(11):555–559, November 2002.

[183] Douglas B. Kell. 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, 2002.

[184] Douglas B. Kell, Robert M. Darby, and John Draper. Genomic computing. explanatory analysis of plant expression profiling data using machine learning. *Plant Physiology*, 126(3):943–951, July 2001.

[185] R. E. Keller and R. Poli. Linear genetic programming of metaheuristics. In Dirk Thierens, Hans-Georg Beyer, Josh Bongard, Jurgen Branke, John Andrew Clark, Dave Cliff, Clare Bates Congdon, Kalyanmoy Deb, Benjamin Doerr, Tim Kovacs, Sanjeev Kumar, Julian F. Miller, Jason Moore, Frank Neumann, Martin Pelikan, Riccardo Poli, Kumara Sastry, Kenneth Owen Stanley, Thomas Stutzle, Richard A Watson, and Ingo Wegener, editors, *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 2, pages 1753–1753, London, 7-11 July 2007. ACM Press.

[186] Robert E. Keller and Riccardo Poli. Cost-benefit investigation of a genetic-programming hyperheuristic. In *Proceedings of Evolution Artificielle*, October 2007.

[187] Robert E. Keller and Riccardo Poli. Linear genetic programming of parsimonious metaheuristics. In *Proceedings of IEEE Congress on Evolutionary Computation (CEC)*, September 2007.

[188] Didier Keymeulen, Adrian Stoica, Jason Lohn, and Ricardo S. Zebulum, editors. *The Third NASA/DoD workshop on Evolvable Hardware*, Long Beach, California, 12-14 July 2001. IEEE Computer Society.

[189] Bijan KHosraviani. Organization design optimization using genetic programming. In John R. Koza, editor, *Genetic Algorithms and Genetic Programming at Stanford 2003*, pages 109–117. Stanford Bookstore, Stanford, California, 94305-3079 USA, 4 December 2003.

[190] Bijan KHosraviani, Raymond E. Levitt, and John R. Koza. Organization design optimization using genetic programming. In Maarten Keijzer, editor, *Late Breaking Papers at the 2004 Genetic and Evolutionary Computation Conference*, Seattle, Washington, USA, 26 July 2004.

[191] Raihan H. Kibria and You Li. Optimizing the initialization of dynamic decision heuristics in DPLL SAT solvers using genetic programming. In Pierre Collet, Marco Tomassini, Marc Ebner, Steven Gustafson, and Anikó Ekárt, editors, *Proceedings of the 9th European Conference on Genetic Programming*, volume 3905 of *Lecture Notes in Computer Science*, pages 331–340, Budapest, Hungary, 10 - 12 April 2006. Springer.

[192] Kenneth E. Kinnear, Jr. Evolving a sort: Lessons in genetic programming. In *Proceedings of the 1993 International Conference on Neural Networks*, volume 2, pages 881–888, San Francisco, USA, 28 March-1 April 1993. IEEE Press.

[193] Kenneth E. Kinnear, Jr., editor. *Advances in Genetic Programming*. MIT Press, Cambridge, MA, 1994.

[194] Kenneth E. Kinnear, Jr. Fitness landscapes and difficulty in genetic programming. In *Proceedings of the 1994 IEEE World Conference on Computational Intelligence*, volume 1, pages 142–147, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.

[195] Kenneth E. Kinnear, Jr. A perspective on the work in this book. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 1, pages 3–19. MIT Press, 1994.

[196] Tim J. Klassen and Malcolm I. Heywood. Towards the on-line recognition of arabic characters. In *Proceedings of the 2002 International Joint Conference on Neural Networks IJCNN'02*, pages 1900–1905, Hilton Hawaiian Village Hotel, Honolulu, Hawaii, 12-17 May 2002. IEEE Press.

[197] Arthur Kordon. Evolutionary computation at dow chemical. *SIGEVOlution*, 1(3):4–9, September 2006.

[198] Arthur Kordon, Flor Castillo, Guido Smits, and Mark Kotanchek. Application issues of genetic programming in industry. In Tina Yu, Rick L. Riolo, and Bill Worzel, editors, *Genetic Programming Theory and Practice III*, volume 9 of *Genetic Programming*, chapter 16, pages 241–258. Springer, Ann Arbor, 12-14 May 2005.

[199] Miha Kovacic and Joze Balic. Evolutionary programming of a CNC cutting machine. *International journal for advanced manufacturing technology*, 22(1-2):118–124, September 2003.

[200] J. Koza. Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems. Technical Report STAN-CS-90-1314, Dept. of Computer Science, Stanford University, June 1990.

[201] J. R. Koza. Hierarchical genetic algorithms operating on populations of computer programs. In N. S. Sridharan, editor, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence IJCAI-89*, volume 1, pages 768–774, Detroit, MI, USA, 20-25 August 1989. Morgan Kaufmann.

[202] John R. Koza. A genetic approach to econometric modeling. In *Sixth World Congress of the Econometric Society*, Barcelona, Spain, 1990.

[203] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[204] John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge Massachusetts, May 1994.

[205] John R. Koza. *Genetic Programming II Videotape: The next generation*. MIT Press, 55 Hayward Street, Cambridge, MA, USA, 1994.

[206] John R. Koza, editor. *Late Breaking Papers at the Genetic Programming 1996 Conference Stanford University July 28-31, 1996*, Stanford University, CA, USA, 28–31 July 1996. Stanford Bookstore.

[207] John R. Koza, editor. *Late Breaking Papers at the 1997 Genetic Programming Conference*, Stanford University, CA, USA, 13–16 July 1997. Stanford Bookstore.

[208] John R. Koza, editor. *Late Breaking Papers at the 1998 Genetic Programming Conference*, University of Wisconsin, Madison, WI, USA, 22-25 July 1998. Omni Press.

[209] John R. Koza, Sameer H. Al-Sakran, and Lee W. Jones. Automated re-invention of six patented optical lens systems using genetic programming. In Hans-Georg Beyer, Una-May O'Reilly, Dirk V. Arnold, Wolfgang Banzhaf, Christian Blum, Eric W. Bonabeau, Erick Cantu-Paz, Dipankar Dasgupta, Kalyanmoy Deb, James A. Foster, Edwin D. de Jong, Hod Lipson, Xavier Llora, Spiros Mancoridis, Martin Pelikan, Guenther R. Raidl, Terence Soule, Andy M. Tyrrell, Jean-Paul Watson, and Eckart Zitzler, editors, *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, volume 2, pages 1953–1960, Washington DC, USA, 25-29 June 2005. ACM Press.

[210] John R. Koza and David Andre. Classifying protein segments as transmembrane domains using architecture-altering operations in genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 8, pages 155–176. MIT Press, Cambridge, MA, USA, 1996.

[211] John R. Koza, David Andre, Forrest H Bennett III, and Martin Keane. *Genetic Programming 3: Darwinian Invention and Problem Solving*. Morgan Kaufman, April 1999.

[212] John R. Koza, David Andre, Forrest H Bennett III, and Martin A. Keane. Use of automatically defined functions and architecture-altering operations in automated circuit synthesis using genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 132–149, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[213] John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors. *Genetic Programming 1998: Proceedings of the Third Annual Conference*, University of Wisconsin, Madison, WI, USA, 22-25 July 1998. Morgan Kaufmann.

[214] John R. Koza, Forrest H Bennett III, David Andre, and Martin A. Keane. Automated WYWIWYG design of both the topology and component values of electrical circuits using genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 123–131, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[215] John R. Koza, Forrest H Bennett III, and Oscar Stiffelman. Genetic programming as a Darwinian invention machine. In Riccardo Poli, Peter Nordin, William B. Langdon, and Terence C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'99*, volume 1598 of *LNCS*, pages 93–108, Goteborg, Sweden, 26-27 May 1999. Springer-Verlag.

[216] John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors. *Genetic Programming 1997: Proceedings of the Second Annual Conference*, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[217] John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors. *Genetic Programming 1996: Proceedings of the First Annual Conference*, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[218] John R. Koza, Martin A. Keane, Matthew J. Streeter, William Mydlowec, Jessen Yu, and Guido Lanza. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers, 2003.

[219] John R. Koza and Riccardo Poli. Genetic programming. In Edmund K. Burke and Graham Kendall, editors, *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, chapter 5. Springer, 2005.

[220] N. Krasnogor. Self generating metaheuristics in bioinformatics: The proteins structure comparison case. *Genetic Programming and Evolvable Machines*, 5(2):181–201, June 2004.

[221] Krzysztof Krawiec. *Evolutionary Feature Programming: Cooperative learning for knowledge discovery and computer vision.* Number 385. Wydawnictwo Politechniki Poznanskiej, Poznan University of Technology, Poznan, Poland, 2004.

[222] W. B. Langdon. The evolution of size in variable length representations. In *1998 IEEE International Conference on Evolutionary Computation*, pages 633–638, Anchorage, Alaska, USA, 5-9 May 1998. IEEE Press.

[223] W. B. Langdon. Scaling of program tree fitness spaces. *Evolutionary Computation*, 7(4):399–428, Winter 1999.

[224] W. B. Langdon. Size fair and homologous tree genetic programming crossovers. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1092–1097, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[225] W. B. Langdon. Convergence rates for the distribution of program outputs. In W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 812–819, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[226] W. B. Langdon. How many good programs are there? How long are they? In Kenneth A. De Jong, Riccardo Poli, and Jonathan E. Rowe, editors, *Foundations of Genetic Algorithms VII*, pages 183–202, Torremolinos, Spain, 4-6 September 2002. Morgan Kaufmann. Published 2003.

[227] W. B. Langdon. Convergence of program fitness landscapes. In E. Cantú-Paz, J. A. Foster, K. Deb, D. Davis, R. Roy, U.-M. O'Reilly, H.-G. Beyer, R. Standish, G. Kendall, S. Wilson, M. Harman, J. Wegener, D. Dasgupta, M. A. Potter, A. C. Schultz, K. Dowsland, N. Jonoska, and J. Miller, editors, *Genetic and Evolutionary Computation – GECCO-2003*, volume 2724 of *LNCS*, pages 1702–1714, Chicago, 12-16 July 2003. Springer-Verlag.

[228] W. B. Langdon. The distribution of reversible functions is Normal. In Rick L. Riolo and Bill Worzel, editors, *Genetic Programming Theory and Practise*, chapter 11, pages 173–188. Kluwer, 2003.

[229] W. B. Langdon. Global distributed evolution of L-systems fractals. In Maarten Keijzer, Una-May O'Reilly, Simon M. Lucas, Ernesto Costa, and Terence Soule, editors, *Genetic Programming, Proceedings of EuroGP'2004*, volume 3003 of *LNCS*, pages 349–358, Coimbra, Portugal, 5-7 April 2004. Springer-Verlag.

[230] W. B. Langdon. The distribution of amorphous computer outputs. In Susan Stepney and Stephen Emmott, editors, *The Grand Challenge in Non-Classical Computation: International Workshop*, York, UK, 18-19 April 2005.

[231] W. B. Langdon. Pfeiffer – A distributed open-ended evolutionary system. In Bruce Edmonds, Nigel Gilbert, Steven Gustafson, David Hales, and Natalio Krasnogor, editors, *AISB'05: Proceedings of the Joint Symposium on Socially Inspired Computing (METAS 2005)*, pages 7–13, University of Hertfordshire, Hatfield, UK, 12-15 April 2005. SSAISB 2005 Convention.

[232] W. B. Langdon. Mapping non-conventional extensions of genetic programming. In Cristian S. Calude, Michael J. Dinneen, Gheorghe Paun, Grzegorz Rozenberg, and Susan Stepney, editors, *Unconventional Computing 2006*, volume 4135 of *LNCS*, pages 166–180, York, 4-8 September 2006. Springer-Verlag.

[233] W. B. Langdon and W. Banzhaf. A SIMD interpreter for genetic programming on GPU graphics cards. In preparation, 3 July 2007.

[234] W. B. Langdon and B. F. Buxton. Genetic programming for mining DNA chip data from cancer patients. *Genetic Programming and Evolvable Machines*, 5(3):251–257, September 2004.

[235] W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, editors. *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[236] W. B. Langdon and A. P. Harrison. GP on SPMD parallel graphics hardware for mega bioinformatics data mining. In preparation.

[237] W. B. Langdon and R. Poli. Fitness causes bloat. In P. K. Chawdhry, R. Roy, and R. K. Pant, editors, *Soft Computing in Engineering Design and Manufacturing*, pages 13–22. Springer-Verlag London, 23-27 June 1997.

[238] W. B. Langdon and R. Poli. The halting probability in von Neumann architectures. In Pierre Collet, Marco Tomassini, Marc Ebner, Steven Gustafson, and Anikó Ekárt, editors, *Proceedings of the 9th European Conference on Genetic Programming*, volume 3905 of *Lecture Notes in Computer Science*, pages 225–237, Budapest, Hungary, 10 - 12 April 2006. Springer.

[239] W. B. Langdon and Riccardo Poli. *Foundations of Genetic Programming*. Springer-Verlag, 2002.

[240] W. B. Langdon, Riccardo Poli, Peter Nordin, and Terry Fogarty, editors. *Late-Breaking Papers of EuroGP-99*, Goteborg, Sweden, 26-27 May 1999.

[241] William B. Langdon. A bibliography for genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter B, pages 507–532. MIT Press, Cambridge, MA, USA, 1996.

[242] William B. Langdon. *Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming!*, volume 1 of *Genetic Programming*. Kluwer, Boston, 24 April 1998.

[243] William B. Langdon. Size fair and homologous tree genetic programming crossovers. *Genetic Programming and Evolvable Machines*, 1(1/2):95–119, April 2000.

[244] William B. Langdon and Wolfgang Banzhaf. Repeated sequences in linear genetic programming genomes. *Complex Systems*, 15(4):285–306, 2005.

[245] William B. Langdon and Peter Nordin. Evolving hand-eye coordination for a humanoid robot with machine code genetic programming. In Julian F. Miller, Marco Tomassini, Pier Luca Lanzi, Conor Ryan, Andrea G. B. Tettamanzi, and William B. Langdon, editors, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 313–324, Lake Como, Italy, 18-20 April 2001. Springer-Verlag.

[246] William B. Langdon and Riccardo Poli. Mapping non-conventional extensions of genetic programming. *Natural Computing*. Invited contribution to special issue on Unconventional computing.

[247] William B. Langdon and Riccardo Poli. On turing complete T7 and MISC F–4 program fitness landscapes. In Dirk V. Arnold, Thomas Jansen, Michael D. Vose, and Jonathan E. Rowe, editors, *Theory of Evolutionary Algorithms*, number 06061 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 5-10 February 2006. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany. <http://drops.dagstuhl.de/opus/volltexte/2006/595> [date of citation: 2006-01-01].

[248] William B. Langdon, Terry Soule, Riccardo Poli, and James A. Foster. The evolution of size and shape. In Lee Spector, William B. Langdon, Una-May O'Reilly, and Peter J. Angeline, editors, *Advances in Genetic Programming 3*, chapter 8, pages 163–190. MIT Press, Cambridge, MA, USA, June 1999.

[249] Kwong Sak Leung, Kin Hong Lee, and Sin Man Cheang. Genetic parallel programming - evolving linear machine codes on a multiple-ALU processor. In Sazali Yaacob, R. Nagarajan, and Ali Chekima, editors, *Proceedings of International Conference on Artificial Intelligence in Engineering and Technology - ICAIET 2002*, pages 207–213. Universiti Malaysia Sabah, June 2002.

[250] T. L. Lew, A. B. Spencer, F. Scarpa, K. Worden, A. Rutherford, and F. Hemez. Identification of response surface models using genetic programming. *Mechanical Systems and Signal Processing*, 20(8):1819–1831, November 2006.

[251] Daniel R. Lewin, Sivan Lachman-Shalem, and Benyamin Grosman. The role of process system engineering (PSE) in integrated circuit (IC) manufacturing. *Control Engineering Practice*, 15(7):793–802, July 2006. Special Issue on Award Winning Applications, 2005 IFAC World Congress.

[252] Li Li, Wei Jiang, Xia Li, Kathy L. Moser, Zheng Guo, Lei Du, Qiuju Wang, Eric J. Topol, Qing Wang, and Shaoqi Rao. A robust hybrid between genetic algorithm and support vector machine for extracting an optimal feature gene subset. *Genomics*, 85(1):16–23, January 2005.

[253] Ricardo Linden and Amit Bhaya. Evolving fuzzy rules to model gene expression. *Biosystems*, 88(1-2):76–91, March 2007.

[254] Hod Lipson. How to draw a straight line using a GP: Benchmarking evolutionary design against 19th century kinematic synthesis. In Maarten Keijzer, editor, *Late Breaking Papers at the 2004 Genetic and Evolutionary Computation Conference*, Seattle, Washington, USA, 26 July 2004.

[255] Weiguo Liu and Bertil Schmidt. Mapping of hierarchical parallel genetic algorithms for protein folding onto computational grids. *IEICE Transactions on Information and Systems*, E89-D(2):589–596, 2006.

[256] Jason Lohn, Gregory Hornby, and Derek Linden. Evolutionary antenna design for a NASA spacecraft. In Una-May O'Reilly, Tina Yu, Rick L. Riolo, and Bill Worzel, editors, *Genetic Programming Theory and Practice II*, chapter 18, pages 301–315. Springer, Ann Arbor, 13-15 May 2004.

[257] Jason Lohn, Adrian Stoica, and Didier Keymeulen, editors. *The Second NASA/DoD Workshop on Evolvable Hardware*, Palo Alto, California, 13-15 July 2000. IEEE Computer Society.

[258] Jason D. Lohn, Gregory S. Hornby, and Derek S. Linden. Rapid re-evolution of an X-band antenna for NASA's space technology 5 mission. In Tina Yu, Rick L. Riolo, and Bill Worzel, editors, *Genetic Programming Theory and Practice III*, volume 9 of *Genetic Programming*, chapter 5, pages 65–78. Springer, Ann Arbor, 12-14 May 2005.

[259] Jean Louchet. Using an individual evolution strategy for stereovision. *Genetic Programming and Evolvable Machines*, 2(2):101–109, June 2001.

[260] Jean Louchet, Maud Guyon, Marie-Jeanne Lesot, and Amine Boumaza. Dynamic flies: a new pattern recognition tool applied to stereo sequence processing. *Pattern Recognition Letters*, 23(1-3):335–345, January 2002.

[261] Jörn Loviscach and Jennis Meyer-Spradow. Genetic programming of vertex shaders. In M. Chover, H. Hagen, and D. Tost, editors, *Proceedings of EuroMedia 2003*, pages 29–31, 2003.

[262] Sean Luke. Evolving soccerbots: A retrospective. In *Proceedings of the 12th Annual Conference of the Japanese Society for Artificial Intelligence*, 1998.

[263] Sean Luke. Two fast tree-creation algorithms for genetic programming. *IEEE Transactions on Evolutionary Computation*, 4(3):274–283, September 2000.

[264] Eduard Lukschandl, Henrik Borgvall, Lars Nohle, Mats Nordahl, and Peter Nordin. Distributed java bytecode genetic programming. In Riccardo Poli, Wolfgang Banzhaf, William B. Langdon, Julian F. Miller, Peter Nordin, and Terence C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802 of *LNCS*, pages 316–325, Edinburgh, 15-16 April 2000. Springer-Verlag.

[265] Penousal Machado and Juan Romero, editors. *The Art of Artificial Evolution*. Springer, 2008.

[266] Peter Marenbach. Using prior knowledge and obtaining process insight in data based modelling of bioprocesses. *System Analysis Modelling Simulation*, 31:39–59, 1998.

[267] Sheri Markose, Edward Tsang, Hakan Er, and Abdel Salhi. Evolutionary arbitrage for FTSE-100 index options and futures. In *Proceedings of the 2001 Congress on Evolutionary Computation CEC2001*, pages 275–282, COEX, World Trade Center, 159 Samseong-dong, Gangnam-gu, Seoul, Korea, 27-30 May 2001. IEEE Press.

[268] John Paul Marney, D. Miller, Colin Fyfe, and Heather F. E. Tarbert. Risk adjusted returns to technical trading rules: a genetic programming approach. In *7th International Conference of Society of Computational Economics*, Yale, 28-29 June 2001.

[269] Martin C. Martin. Evolving visual sonar: Depth from monocular images. *Pattern Recognition Letters*, 27(11):1174–1180, August 2006. Evolutionary Computer Vision and Image Understanding.

[270] Peter Martin. A hardware implementation of a genetic programming system using FPGAs and Handel-C. *Genetic Programming and Evolvable Machines*, 2(4):317–343, December 2001.

[271] Paul Massey, John A. Clark, and Susan Stepney. Evolution of a human-competitive quantum fourier transform algorithm using genetic programming. In Hans-Georg Beyer, Una-May O'Reilly, Dirk V. Arnold, Wolfgang Banzhaf, Christian Blum, Eric W. Bonabeau, Erick Cantu-Paz, Dipankar Dasgupta, Kalyanmoy Deb, James A. Foster, Edwin D. de Jong, Hod Lipson, Xavier Llora, Spiros Mancoridis, Martin Pelikan, Guenther R. Raidl, Terence Soule, Andy M. Tyrrell, Jean-Paul Watson, and Eckart Zitzler, editors, *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, volume 2, pages 1657–1663, Washington DC, USA, 25-29 June 2005. ACM Press.

[272] Sidney R. Maxwell III. Experiments with a coroutine model for genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 413–417a, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.

[273] Jon McCormack. New challenges for evolutionary music and art. *SIGEvolution*, 1(1):5–11, April 2006.

[274] Aoife C. McGovern, David Broadhurst, Janet Taylor, Naheed Kaderbhai, Michael K. Winson, David A. Small, Jem J. Rowland, Douglas B. Kell, and Royston Goodacre. 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, 5 June 2002.

[275] Ben McKay, Mark Willis, Dominic Searson, and Gary Montague. Nonlinear continuum regression: an evolutionary approach. *Transactions of the Institute of Measurement and Control*, 22(2):125–140, 2000.

[276] Nicholas Freitag McPhee, Nicholas J. Hopper, and Mitchell L. Reierson. Sutherland: An extensible object-oriented software framework for evolutionary computation. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, page 241, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.

[277] Nicholas Freitag McPhee and Justin Darwin Miller. Accurate replication in genetic programming. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 303–309, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.

[278] Peter Kip Mercure, Guido F. Smits, and Arthur Kordon. Empirical emulators for first principle models. In *AIChE Fall Annual Meeting*, Reno Hilton, 6 November 2001.

[279] Jennis Meyer-Spradow and Jörn Loviscach. Evolutionary design of BRDFs. In M. Chover, H. Hagen, and D. Tost, editors, *Eurographics 2003 Short Paper Proceedings*, pages 301–306, 2003.

[280] Julian Miller, Marco Tomassini, Pier Luca Lanzi, Conor Ryan, Andrea G. B. Tettamanzi, and William B. Langdon, editors. *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, Lake Como, Italy, 18-20 April 2001. Springer-Verlag.

[281] Julian F. Miller. An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E.

Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1135–1142, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[282] Julian F. Miller and Stephen L. Smith. Redundancy and computational efficiency in cartesian genetic programming. *IEEE Transactions on Evolutionary Computation*, 10(2):167–174, April 2006.

[283] Julian F. Miller, Adrian Thompson, Peter Thomson, and Terence C. Fogarty, editors. *Proceedings of the Third International Conference on Evolvable Systems, ICES 2000*, volume 1801 of *LNCS*, Edinburgh, Scotland, UK, 17-19 April 2000. Springer-Verlag.

[284] Boris Mitavskiy and Jon Rowe. Some results about the markov chains associated to GPs and to general EAs. *Theoretical Computer Science*, 361(1):72–110, 28 August 2006.

[285] David J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[286] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 1965.

[287] Jason H. Moore, Joel S. Parker, Nancy J. Olsen, and Thomas M. Aune. Symbolic discriminant analysis of microarray data in automimmune disease. *Genetic Epidemiology*, 23:57–69, 2002.

[288] Alison A Motsinger, Stephen L Lee, George Mellick, and Marylyn D Ritchie. 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, January 25 2006.

[289] Christopher J. Neely. Risk-adjusted, ex ante, optimal technical trading rules in equity markets. *International Review of Economics and Finance*, 12(1):69–87, Spring 2003.

[290] Christopher J. Neely and Paul A. Weller. Technical trading rules in the european monetary system. *Journal of International Money and Finance*, 18(3):429–458, 1999.

[291] Christopher J. Neely and Paul A. Weller. 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, September 2001.

[292] Christopher J. Neely and Paul A. Weller. Technical analysis and central bank intervention. *Journal of International Money and Finance*, 20(7):949–970, December 2001.

[293] Christopher J. Neely, Paul A. Weller, and Rob Dittmar. Is technical analysis in the foreign exchange market profitable? A genetic programming approach. *The Journal of Financial and Quantitative Analysis*, 32(4):405–426, December 1997.

[294] Christopher J. Neely, Paul A. Weller, and Joshua M. Ulrich. 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, August 2006. Revised March 2007.

[295] Nikolay Nikolaev and Hitoshi Iba. *Adaptive Learning of Polynomial Networks Genetic Programming, Backpropagation and Bayesian Methods*. Number 4 in Genetic and Evolutionary Computation. Springer, 2006. June.

[296] Nikolay Y. Nikolaev and Hitoshi Iba. Genetic programming of polynomial models for financial forecasting. In Shu-Heng Chen, editor, *Genetic Algorithms and Genetic Programming in Computational Finance*, chapter 5, pages 103–123. Kluwer Academic Press, 2002.

[297] Allen E. Nix and Michael D. Vose. Modeling genetic algorithms with Markov chains. *Annals of Mathematics and Artificial Intelligence*, 5:79–88, 1992.

[298] Peter Nordin. A compiling genetic programming system that directly manipulates the machine code. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 14, pages 311–331. MIT Press, 1994.

[299] Peter Nordin. *Evolutionary Program Induction of Binary Machine Code and its Applications*. PhD thesis, der Universitat Dortmund am Fachereich Informatik, 1997.

[300] Peter Nordin, Wolfgang Banzhaf, and Frank D. Francone. Efficient evolution of machine code for CISC architectures using instruction blocks and homologous crossover. In Lee Spector, William B. Langdon, Una-May O'Reilly, and Peter J. Angeline, editors, *Advances in Genetic Programming 3*, chapter 12, pages 275–299. MIT Press, Cambridge, MA, USA, June 1999.

[301] Peter Nordin and Wilde Johanna. *Humanoider: Sjavlarande robotar och artificiell intelligens*. Liber, 2003.

[302] Howard Oakley. Two scientific applications of genetic programming: Stack filters and non-linear equation fitting to chaotic data. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 17, pages 369–389. MIT Press, 1994.

[303] Mihai Oltean. Evolving evolutionary algorithms using linear genetic programming. *Evolutionary Computation*, 13(3):387–410, Fall 2005.

[304] Mihai Oltean and D. Dumitrescu. Evolving TSP heuristics using multi expression programming. In Marian Bubak, Geert Dick van Albada, Peter M. A. Sloot, and

Jack Dongarra, editors, *Computational Science - ICCS 2004: 4th International Conference, Part II*, volume 3037 of *Lecture Notes in Computer Science*, pages 670–673, Krakow, Poland, 6-9 June 2004. Springer-Verlag.

[305] Michael O'Neill and Conor Ryan. *Grammatical Evolution: Evolutionary Automatic Programming in a Arbitrary Language*, volume 4 of *Genetic programming*. Kluwer Academic Publishers, 2003.

[306] S. Openshaw and I. Turton. Building new spatial interaction models using genetic programming. In T. C. Fogarty, editor, *Evolutionary Computing*, Lecture Notes in Computer Science, Leeds, UK, 11-13 April 1994. Springer-Verlag.

[307] Una-May O'Reilly. *An Analysis of Genetic Programming*. PhD thesis, Carleton University, Ottawa-Carleton Institute for Computer Science, Ottawa, Ontario, Canada, 22 September 1995.

[308] Una-May O'Reilly and Martin Hemberg. Integrating generative growth and evolutionary computation for form exploration. *Genetic Programming and Evolvable Machines*, 8(2):163–186, June 2007. Special issue on developmental systems.

[309] Una-May O'Reilly and Franz Oppacher. The troubling aspects of a building block hypothesis for genetic programming. In L. Darrell Whitley and Michael D. Vose, editors, *Foundations of Genetic Algorithms 3*, pages 73–88, Estes Park, Colorado, USA, 31 July–2 August 1994. Morgan Kaufmann. Published 1995.

[310] Una-May O'Reilly, Tina Yu, Rick L. Riolo, and Bill Worzel, editors. *Genetic Programming Theory and Practice II*, volume 8 of *Genetic Programming*, Ann Arbor, MI, USA, 13-15 May 2004. Springer.

[311] Mouloud Oussaidène, Bastien Chopard, Olivier V. Pictet, and Marco Tomassini. Parallel genetic programming and its application to trading model induction. *Parallel Computing*, 23(8):1183–1198, August 1997.

[312] John D. Owens, David, Naga Govindaraju, Mark Harris, Jens Kruger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, March 2007.

[313] Daniel Parrott, Xiaodong Li, and Vic Ciesielski. Multi-objective techniques in genetic programming for evolving classifiers. In David Corne, Zbigniew Michalewicz, Marco Dorigo, Gusz Eiben, David Fogel, Carlos Fonseca, Garrison Greenwood, Tan Kay Chen, Guenther Raidl, Ali Zalzala, Simon Lucas, Ben Paechter, Jennifier Willies, Juan J. Merelo Guervos, Eugene Eberbach, Bob McKay, Alastair Channon, Ashutosh Tiwari, L. Gwenn Volkert, Dan Ashlock, and Marc Schoenauer, editors, *Proceedings of the 2005 IEEE Congress on Evolutionary Computation*, volume 2, pages 1141–1148, Edinburgh, UK, 2-5 September 2005. IEEE Press.

[314] Tim Perkis. Stack-based genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 148–153, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.

[315] Nelishia Pillay. Evolving solutions to ASCII graphics programming problems in intelligent programming tutors. In Rajendra Akerkar, editor, *International Conference on Applied Artificial Intelligence (ICAAI'2003)*, pages 236–243, Fort Panhala, Kolhapur, India, 15-16 December 2003. TMRF.

[316] R. Poli. Hyperschema theory for GP with one-point crossover, building blocks, and some new results in GA theory. In Riccardo Poli, Wolfgang Banzhaf, William B. Langdon, Julian F. Miller, Peter Nordin, and Terence C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802 of *LNCS*, pages 163–180, Edinburgh, 15-16 April 2000. Springer-Verlag.

[317] Riccardo Poli. Discovery of symbolic, neuro-symbolic and neural networks with parallel distributed genetic programming. Technical Report CSRP-96-14, University of Birmingham, School of Computer Science, August 1996. Presented at 3rd International Conference on Artificial Neural Networks and Genetic Algorithms, ICANNGA'97.

[318] Riccardo Poli. Genetic programming for image analysis. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 363–368, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[319] Riccardo Poli. Parallel distributed genetic programming. In David Corne, Marco Dorigo, and Fred Glover, editors, *New Ideas in Optimization*, Advanced Topics in Computer Science, chapter 27, pages 403–431. McGraw-Hill, Maidenhead, Berkshire, England, 1999.

[320] Riccardo Poli. Exact schema theorem and effective fitness for GP with one-point crossover. In D. Whitley, D. Goldberg, E. Cantu-Paz, L. Spector, I. Parmee, and H.-G. Beyer, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 469–476, Las Vegas, July 2000. Morgan Kaufmann.

[321] Riccardo Poli. Exact schema theory for genetic programming and variable-length genetic algorithms with one-point crossover. *Genetic Programming and Evolvable Machines*, 2(2):123–163, 2001.

[322] Riccardo Poli. A simple but theoretically-motivated method to control bloat in genetic programming. In Conor Ryan, Terence Soule, Maarten Keijzer, Edward Tsang, Riccardo Poli, and Ernesto Costa, editors, *Genetic Programming, Proceedings of EuroGP'2003*, volume 2610 of *LNCS*, pages 204–217, Essex, 14-16 April 2003. Springer-Verlag.

[323] Riccardo Poli. Tournament selection, iterated coupon-collection problem, and backward-chaining evolutionary algorithms. In Alden H. Wright, Michael D. Vose, Kenneth A. De Jong, and Lothar M. Schmitt, editors, *Foundations of Genetic Algorithms 8*, volume 3469 of *Lecture Notes in Computer Science*, pages 132–155, Aizu-Wakamatsu City, Japan, 5-9 January 2005. Springer-Verlag.

[324] Riccardo Poli, Wolfgang Banzhaf, William B. Langdon, Julian F. Miller, Peter Nordin, and Terence C. Fogarty, editors. *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802 of *LNCS*, Edinburgh, 15-16 April 2000. Springer-Verlag.

[325] Riccardo Poli, Cecilia Di Chio, and William B. Langdon. Exploring extended particle swarms: a genetic programming approach. In Hans-Georg Beyer, Una-May O'Reilly, Dirk V. Arnold, Wolfgang Banzhaf, Christian Blum, Eric W. Bonabeau, Erick Cantu-Paz, Dipankar Dasgupta, Kalyanmoy Deb, James A. Foster, Edwin D. de Jong, Hod Lipson, Xavier Llora, Spiros Mancoridis, Martin Pelikan, Guenther R. Raidl, Terence Soule, Andy M. Tyrrell, Jean-Paul Watson, and Eckart Zitzler, editors, *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, volume 1, pages 169–176, Washington DC, USA, 25-29 June 2005. ACM Press.

[326] Riccardo Poli and W. B. Langdon. A new schema theory for genetic programming with one-point crossover and point mutation. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 278–285, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[327] Riccardo Poli, W. B. Langdon, Marc Schoenauer, Terry Fogarty, and Wolfgang Banzhaf, editors. *Late Breaking Papers at EuroGP'98: the First European Workshop on Genetic Programming*, Paris, France, 14-15 April 1998.

[328] Riccardo Poli and William B. Langdon. On the search properties of different crossover operators in genetic programming. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 293–301, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.

[329] Riccardo Poli and William B. Langdon. Schema theory for genetic programming with one-point crossover and point mutation. *Evolutionary Computation*, 6(3):231–252, 1998.

[330] Riccardo Poli and William B. Langdon. Running genetic programming backward. In Rick L. Riolo, Bill Worzel, and Tina Yu, editors, *Genetic Programming Theory and Practice*. Kluwer, 2005.

[331] Riccardo Poli and William B. Langdon. Running genetic programming backward. In Tina Yu, Rick L. Riolo, and Bill Worzel, editors, *Genetic Programming Theory and*

Practice III, volume 9 of *Genetic Programming*, chapter 9, pages 125–140. Springer, Ann Arbor, 12-14 May 2005.

[332] Riccardo Poli and William B. Langdon. Backward-chaining evolutionary algorithms. *Artificial Intelligence*, 170(11):953–982, August 2006.

[333] Riccardo Poli and William B. Langdon. Efficient markov chain model of machine code program execution and halting. In Rick L. Riolo, Terence Soule, and Bill Worzel, editors, *Genetic Programming Theory and Practice IV*, volume 5 of *Genetic and Evolutionary Computation*, chapter 13. Springer, Ann Arbor, 11-13 May 2006.

[334] Riccardo Poli, William B. Langdon, and Stephen Dignum. On the limiting distribution of program sizes in tree-based genetic programming. In Marc Ebner, Michael O'Neill, Anikó Ekárt, Leonardo Vanneschi, and Anna Isabel Esparcia-Alcázar, editors, *Proceedings of the 10th European Conference on Genetic Programming*, volume 4445 of *Lecture Notes in Computer Science*, pages 193–204, Valencia, Spain, 11 - 13 April 2007. Springer.

[335] Riccardo Poli, William B. Langdon, and Owen Holland. Extending particle swarm optimisation via genetic programming. In Maarten Keijzer, Andrea Tettamanzi, Pierre Collet, Jano I. van Hemert, and Marco Tomassini, editors, *Proceedings of the 8th European Conference on Genetic Programming*, volume 3447 of *Lecture Notes in Computer Science*, pages 291–300, Lausanne, Switzerland, 30 March - 1 April 2005. Springer.

[336] Riccardo Poli and Nicholas Freitag McPhee. General schema theory for genetic programming with subtree-swapping crossover: Part I. *Evolutionary Computation*, 11(1):53–66, March 2003.

[337] Riccardo Poli and Nicholas Freitag McPhee. General schema theory for genetic programming with subtree-swapping crossover: Part II. *Evolutionary Computation*, 11(2):169–206, June 2003.

[338] Riccardo Poli, Nicholas Freitag McPhee, and Jonathan E. Rowe. 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, March 2004.

[339] Riccardo Poli, Peter Nordin, William B. Langdon, and Terence C. Fogarty, editors. *Genetic Programming, Proceedings of EuroGP'99*, volume 1598 of *LNCS*, Goteborg, Sweden, 26-27 May 1999. Springer-Verlag.

[340] Riccardo Poli, Jonathan Page, and W. B. Langdon. Smooth uniform crossover, submachine code GP and demes: A recipe for solving high-order boolean parity problems. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic*

and Evolutionary Computation Conference, volume 2, pages 1162–1169, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[341] Riccardo Poli, Jonathan E. Rowe, and Nicholas Freitag McPhee. Markov chain models for GP and variable-length GAs with homologous crossover. In Lee Spector, Erik D. Goodman, Annie Wu, W. B. Langdon, Hans-Michael Voigt, Mitsuo Gen, Sandip Sen, Marco Dorigo, Shahram Pezeshk, Max H. Garzon, and Edmund Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 112–119, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann.

[342] Riccardo Poli, John Woodward, and Edmund Burke. A histogram-matching approach to the evolution of bin-packing strategies. In *Proceedings of the IEEE Congress on Evolutionary Computation*, Singapore, 2007. accepted.

[343] Mitchell A. Potter. *The Design and Analysis of a Computational Model of Cooperative Coevolution*. PhD thesis, George Mason University, Washington, DC, spring 1997.

[344] S. Priesterjahn, O. Kramer, A. Weimer, and A. Goebels. Evolution of human-competitive agents in modern computer games. In Gary G. Yen, Simon M. Lucas, Gary Fogel, Graham Kendall, Ralf Salomon, Byoung-Tak Zhang, Carlos A. Coello Coello, and Thomas Philip Runarsson, editors, *Proceedings of the 2006 IEEE Congress on Evolutionary Computation*, pages 777–784, Vancouver, BC, Canada, 16-21 July 2006. IEEE Press.

[345] Adam Prügel-Bennett and Jonathan L. Shapiro. An analysis of genetic algorithms using statistical mechanics. *Physical Review Letters*, 72:1305–1309, 1994.

[346] Marcos I. Quintana, Riccardo Poli, and Ela Claridge. Morphological algorithm design for binary images using genetic programming. *Genetic Programming and Evolvable Machines*, 7(1):81–102, March 2006.

[347] Alain Ratle and Michele Sebag. Genetic programming and domain knowledge: Beyond the limitations of grammar-guided machine discovery. In Marc Schoenauer, Kalyanmoy Deb, Günter Rudolph, Xin Yao, Evelyne Lutton, Juan Julian Merelo, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature - PPSN VI 6th International Conference*, volume 1917 of *LNCS*, pages 211–220, Paris, France, 16-20 September 2000. Springer Verlag.

[348] J. Reggia, M. Tagamets, J. Contreras-Vidal, D. Jacobs, S. Weems, W. Naqvi, R. Winder, T. Chabuk, J. Jung, and C. Yang. Development of a large-scale integrated neurocognitive architecture - part 2: Design and architecture. Technical Report TR-CS-4827, UMIACS-TR-2006-43, University of Maryland, USA, October 2006.

[349] David M. Reif, Bill C. White, and Jason H. Moore. Integrated analysis of genetic, genomic, and proteomic data. *Expert Review of Proteomics*, 1(1):67–75, 2004.

[350] Craig W. Reynolds. Flocks, herds, and schools: A distributed behavioral model. *SIGGRAPH Computer Graphics*, 21(4):25–34, July 1987.

[351] Rick L. Riolo, Terence Soule, and Bill Worzel, editors. *Genetic Programming Theory and Practice IV*, volume 5 of *Genetic and Evolutionary Computation*, Ann Arbor, 11-13 May 2007. Springer.

[352] Rick L. Riolo, Terence Soule, and Bill Worzel, editors. *Genetic Programming Theory and Practice V*, Genetic and Evolutionary Computation, Ann Arbor, 17-19 May 2007. Springer.

[353] Rick L. Riolo and Bill Worzel. *Genetic Programming Theory and Practice*, volume 6 of *Genetic Programming*. Kluwer, Boston, MA, USA, 2003. Series Editor - John Koza.

[354] Marylyn D. Ritchie, Alison A. Motsinger, William S. Bush, Christopher S. Coffey, and Jason H. Moore. Genetic programming neural networks: A powerful bioinformatics tool for human genetics. *Applied Soft Computing*, 7(1):471–479, January 2007.

[355] Marylyn D. Ritchie, Bill C. White, Joel S. Parker, Lance W. Hahn, and Jason H. Moore. 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), 7 July 2003.

[356] Daniel Rivero, Juan R. Rabu nal, Julián Dorado, and Alejandro Pazos. Using genetic programming for character discrimination in damaged documents. In Guenther R. Raidl, Stefano Cagnoni, Jurgen Branke, David W. Corne, Rolf Drechsler, Yaochu Jin, Colin R. Johnson, Penousal Machado, Elena Marchiori, Franz Rothlauf, George D. Smith, and Giovanni Squillero, editors, *Applications of Evolutionary Computing, EvoWorkshops2004: EvoBIO, EvoCOMNET, EvoHOT, EvoIASP, EvoMUSART, EvoSTOC*, volume 3005 of *LNCS*, pages 349–358, Coimbra, Portugal, 5-7 April 2004. Springer Verlag.

[357] Katya Rodriguez-Vazquez, Carlos M. Fonseca, and Peter J. Fleming. Identifying the structure of nonlinear dynamic systems using multiobjective genetic programming. *IEEE Transactions on Systems, Man and Cybernetics, Part A*, 34(4):531–545, July 2004.

[358] Justinian P. Rosca. Analysis of complexity drift in genetic programming. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second*

Annual Conference, pages 286–294, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[359] Justinian P. Rosca and Dana H. Ballard. Discovery of subroutines in genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 9, pages 177–202. MIT Press, Cambridge, MA, USA, 1996.

[360] Brian J. Ross, Anthony G. Gualtieri, Frank Fueten, and Paul Budkewitsch. Hyperspectral image analysis using genetic programming. *Applied Soft Computing*, 5(2):147–156, January 2005.

[361] Franz Rothlauf. *Representations for genetic and evolutionary algorithms*. Springer-Verlag, pub-SV:adr, second edition, 2006. First published 2002, 2nd edition available electronically.

[362] Conor Ryan. *Automatic Re-engineering of Software Using Genetic Programming*, volume 2 of *Genetic Programming*. Kluwer Academic Publishers, 1 November 1999.

[363] Conor Ryan, J. J. Collins, and Michael O'Neill. Grammatical evolution: Evolving programs for an arbitrary language. In Wolfgang Banzhaf, Riccardo Poli, Marc Schoenauer, and Terence C. Fogarty, editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391 of *LNCS*, pages 83–95, Paris, 14-15 April 1998. Springer-Verlag.

[364] Conor Ryan and Laur Ivan. An automatice software re-engineering tool based on genetic programming. In Lee Spector, William B. Langdon, Una-May O'Reilly, and Peter J. Angeline, editors, *Advances in Genetic Programming 3*, chapter 2, pages 15–39. MIT Press, Cambridge, MA, USA, June 1999.

[365] Conor Ryan, Terence Soule, Maarten Keijzer, Edward Tsang, Riccardo Poli, and Ernesto Costa, editors. *Genetic Programming, Proceedings of the 6th European Conference, EuroGP 2003*, volume 2610 of *LNCS*, Essex, UK, 14-16 April 2003. Springer-Verlag.

[366] Arthur L. Samuel. AI, where it has been and where it is going. In *IJCAI*, pages 1152–1157, 1983.

[367] Michael D. Schmidt and Hod Lipson. Co-evolving fitness predictors for accelerating and reducing evaluations. In Rick L. Riolo, Terence Soule, and Bill Worzel, editors, *Genetic Programming Theory and Practice IV*, volume 5 of *Genetic and Evolutionary Computation*, chapter 17, pages –. Springer, Ann Arbor, 11-13 May 2006.

[368] Marc Schoenauer, Kalyanmoy Deb, Günter Rudolph, Xin Yao, Evelyne Lutton, Juan Julian Merelo, and Hans-Paul Schwefel, editors. *Parallel Problem Solving from Nature - PPSN VI 6th International Conference*, volume 1917 of *LNCS*, Paris, France, 16-20 September 2000. Springer Verlag.

[369] Marc Schoenauer, Bertrand Lamy, and Francois Jouve. Identification of mechanical behaviour by genetic programming part II: Energy formulation. Technical report, Ecole Polytechnique, 91128 Palaiseau, France, 1995.

[370] Marc Schoenauer and Michele Sebag. Using domain knowledge in evolutionary system identification. In K. C. Giannakoglou, D. Tsahalis, J. Periaux, K. Papailiou, and T. C. Fogarty, editors, *Evolutionary Methods for Design, Optimization and Control with Applications to Industrial Problems*, Athens, 19-21 September 2001.

[371] Marc Schoenauer, Michele Sebag, Francois Jouve, Bertrand Lamy, and Habibou Maitournam. Evolutionary identification of macro-mechanical models. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 23, pages 467–488. MIT Press, Cambridge, MA, USA, 1996.

[372] D. P. Searson, G. A. Montague, and M. J Willis. 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)*, volume 455 of *IEE Conference Publications*, University of Wales, Swansea, UK, 1-4 September 1998. Institution of Electrical Engineers (IEE).

[373] Lukas Sekanina. *Evolvable Components: From Theory to Hardware Implementations*. Natural Computing. Springer-Verlag, 2003.

[374] Christian Setzkorn. *On The Use Of Multi-Objective Evolutionary Algorithms For Classification Rule Induction*. PhD thesis, University of Liverpool, UK, March 2005.

[375] Shital C. Shah and Andrew Kusiak. Data mining and genetic algorithm based gene/SNP selection. *Artificial Intelligence in Medicine*, 31(3):183–196, July 2004.

[376] Shai Sharabi and Moshe Sipper. GP-sumo: Using genetic programming to evolve sumobots. *Genetic Programming and Evolvable Machines*, 7(3):211–230, October 2006.

[377] Ken C. Sharman and Anna I. Esparcia-Alcazar. Genetic evolution of symbolic signal models. In *Proceedings of the Second International Conference on Natural Algorithms in Signal Processing, NASP'93*, Essex University, UK, 15-16 November 1993.

[378] Ken C. Sharman, Anna I. Esparcia Alcazar, and Yun Li. Evolving signal processing algorithms by genetic programming. In A. M. S. Zalzala, editor, *First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications, GALESIA*, volume 414, pages 473–480, Sheffield, UK, 12-14 September 1995. IEE.

[379] A. D. Shaw, M. K. Winson, A. M. Woodward, A. C. McGovern, H. M. Davey, N. Kaderbhai, D. Broadhurst, R. J. Gilbert, J. Taylor, E. M. Timmins, R. Goodacre,

D. B. Kell, B. K. Alsberg, and J. J. Rowland. 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, January 2000.

[380] Yehonatan Shichel, Eran Ziserman, and Moshe Sipper. GP-robocode: Using genetic programming to evolve robocode players. In Maarten Keijzer, Andrea Tettamanzi, Pierre Collet, Jano I. van Hemert, and Marco Tomassini, editors, *Proceedings of the 8th European Conference on Genetic Programming*, volume 3447 of *Lecture Notes in Computer Science*, pages 143–154, Lausanne, Switzerland, 30 March - 1 April 2005. Springer.

[381] Hong Zong Si, Tao Wang, Ke Jun Zhang, Zhi De Hu, and Bo Tao Fan. QSAR study of 1,4-dihydropyridine calcium channel antagonists based on gene expression programming. *Bioorganic & Medicinal Chemistry*, 14(14):4834–4841, 15 July 2006.

[382] Eric V. Siegel. Competitively evolving decision trees against fixed training cases for natural language processing. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 19, pages 409–423. MIT Press, 1994.

[383] Karl Sims. Artificial evolution for computer graphics. *ACM Computer Graphics*, 25(4):319–328, July 1991. SIGGRAPH '91 Proceedings.

[384] Will Smart and Mengjie Zhang. Applying online gradient descent search to genetic programming for object recognition. In James Hogan, Paul Montague, Martin Purvis, and Chris Steketee, editors, *CRPIT '04: Proceedings of the second workshop on Australasian information security, Data Mining and Web Intelligence, and Software Internationalisation*, volume 32 no. 7, pages 133–138, Dunedin, New Zealand, January 2004. Australian Computer Society, Inc.

[385] Terence Soule and James A. Foster. Effects of code growth and parsimony pressure on populations in genetic programming. *Evolutionary Computation*, 6(4):293–309, Winter 1998.

[386] Terence Soule and James A. Foster. Removal bias: a new cause of code growth in tree based evolutionary programming. In *1998 IEEE International Conference on Evolutionary Computation*, pages 781–186, Anchorage, Alaska, USA, 5-9 May 1998. IEEE Press.

[387] Lee Spector. *Automatic Quantum Computer Programming: A Genetic Programming Approach*, volume 7 of *Genetic Programming*. Kluwer Academic Publishers, Boston/Dordrecht/New York/London, June 2004.

[388] Lee Spector and Adam Alpern. Criticism, culture, and the automatic generation of artworks. In *Proceedings of Twelfth National Conference on Artificial Intelligence*, pages 3–8, Seattle, Washington, USA, 1994. AAAI Press/MIT Press.

[389] Lee Spector and Adam Alpern. 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, 20-25 August 1995.

[390] Lee Spector, Howard Barnum, and Herbert J. Bernstein. Genetic programming for quantum computers. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 365–373, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.

[391] Lee Spector, Howard Barnum, Herbert J. Bernstein, and Nikhil Swamy. Finding a better-than-classical quantum AND/OR algorithm using genetic programming. In Peter J. Angeline, Zbyszek Michalewicz, Marc Schoenauer, Xin Yao, and Ali Zalzala, editors, *Proceedings of the Congress on Evolutionary Computation*, volume 3, pages 2239–2246, Mayflower Hotel, Washington D.C., USA, 6-9 July 1999. IEEE Press.

[392] Lee Spector, Erik D. Goodman, Annie Wu, W. B. Langdon, Hans-Michael Voigt, Mitsuo Gen, Sandip Sen, Marco Dorigo, Shahram Pezeshk, Max H. Garzon, and Edmund Burke, editors. *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-2001*, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann.

[393] Lee Spector, W. B. Langdon, Una-May O'Reilly, and Peter J. Angeline, editors. *Advances in Genetic Programming 3*. MIT Press, Cambridge, MA, USA, June 1999.

[394] Joachim Stender, editor. *Parallel Genetic Algorithms: Theory and Applications*. IOS press, 1993.

[395] C. R. Stephens and H. Waelbroeck. Effective degrees of freedom in genetic algorithms and the block hypothesis. In Thomas Bäck, editor, *Proceedings of the Seventh International Conference on Genetic Algorithms (ICGA97)*, pages 34–40, East Lansing, 1997. Morgan Kaufmann.

[396] C. R. Stephens and H. Waelbroeck. Schemata evolution and building blocks. *Evolutionary Computation*, 7(2):109–124, 1999.

[397] Thomas Sterling. Beowulf-class clustered computing: Harnessing the power of parallelism in a pile of PCs. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, page 883, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann. Invited talk.

[398] Adrian Stoica, Jason Lohn, and Didier Keymeulen, editors. *The First NASA/DoD Workshop on Evolvable Hardware*, Pasadena, California, 19-21 July 1999. IEEE Computer Society.

[399] John J. Szymanski, Steven P. Brumby, Paul Pope, Damian Eads, Diana Esch-Mosher, Mark Galassi, Neal R. Harvey, Hersey D. W. McCulloch, Simon J. Perkins, Reid Porter, James Theiler, A. Cody Young, Jeffrey J. Bloch, and Nancy David. Feature extraction from multiple data sources using genetic programming. In Sylvia S. Shen and Paul E. Lewis, editors, *Algorithms and Technologies for Multispectral, Hyperspectral, and Ultraspectral Imagery VIII*, volume 4725 of *SPIE*, pages 338–345, August 2002.

[400] Walter Alden Tackett. Genetic generation of "dendritic" trees for image classification. In *Proceedings of WCNN93*, pages IV 646–649. IEEE Press, July 1993.

[401] Hideyuki Takagi. Interactive evolutionary computation: Fusion of the capabilities of EC optimization and human evaluation. *Proceedings of the IEEE*, 89(9):1275–1296, September 2001. Invited Paper.

[402] Ivan Tanev, Takashi Uozumi, and Dauren Akhmetov. Component object based single system image for dependable implementation of genetic programming on clusters. *Cluster Computing Journal*, 7(4):347–356, October 2004.

[403] Janet Taylor, Royston Goodacre, William G. Wade, Jem J. Rowland, and Douglas B. Kell. The deconvolution of pyrolysis mass spectra using genetic programming: application to the identification of some eubacterium species. *FEMS Microbiology Letters*, 160:237–246, 1998.

[404] Astro Teller. Genetic programming, indexed memory, the halting problem, and other curiosities. In *Proceedings of the 7th annual Florida Artificial Intelligence Research Symposium*, pages 270–274, Pensacola, Florida, USA, May 1994. IEEE Press.

[405] Astro Teller. Evolving programmers: The co-evolution of intelligent recombination operators. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 3, pages 45–68. MIT Press, Cambridge, MA, USA, 1996.

[406] Astro Teller and David Andre. Automatically choosing the number of fitness cases: The rational allocation of trials. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 321–328, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[407] Ankur Teredesai and Venu Govindaraju. GP-based secondary classifiers. *Pattern Recognition*, 38(4):505–512, April 2005.

[408] James P. Theiler, Neal R. Harvey, Steven P. Brumby, John J. Szymanski, Steve Alferink, Simon J. Perkins, Reid B. Porter, and Jeffrey J. Bloch. Evolving retrieval algorithms with a genetic programming scheme. In Michael R. Descour and Sylvia S. Shen, editors, *Proceedings of SPIE 3753 Imaging Spectrometry V*, pages 416–425, 1999.

[409] Dirk Thierens, Hans-Georg Beyer, Josh Bongard, Jurgen Branke, John Andrew Clark, Dave Cliff, Clare Bates Congdon, Kalyanmoy Deb, Benjamin Doerr, Tim Kovacs, Sanjeev Kumar, Julian F. Miller, Jason Moore, Frank Neumann, Martin Pelikan, Riccardo Poli, Kumara Sastry, Kenneth Owen Stanley, Thomas Stutzle, Richard A Watson, and Ingo Wegener, editors. *GECCO 2007: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, London, UK, 7-11 July 2007. ACM Press.

[410] Peter M. Todd and Gregory M. Werner. Frankensteinian approaches to evolutionary music composition. In Niall Griffith and Peter M. Todd, editors, *Musical Networks: Parallel Distributed Perception and Performance*, pages 313–340. MIT Press, 1999.

[411] Marco Tomassini, L. Luthi, M. Giacobini, and W. B. Langdon. The structure of the genetic programming collaboration network. *Genetic Programming and Evolvable Machines*, 8(1):97–103, March 2007.

[412] L. Trujillo and G. Olague. Using evolution to learn how to perform interest point detection. In X. Y Tang et al., editor, *ICPR 2006 18th International Conference on Pattern Recognition*, volume 1, pages 211–214. IEEE, 20-24 August 2006.

[413] Leonardo Trujillo and Gustavo Olague. Synthesis of interest point detectors through genetic programming. In Maarten Keijzer, Mike Cattolico, Dirk Arnold, Vladan Babovic, Christian Blum, Peter Bosman, Martin V. Butz, Carlos Coello Coello, Dipankar Dasgupta, Sevan G. Ficici, James Foster, Arturo Hernandez-Aguirre, Greg Hornby, Hod Lipson, Phil McMinn, Jason Moore, Guenther Raidl, Franz Rothlauf, Conor Ryan, and Dirk Thierens, editors, *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, volume 1, pages 887–894, Seattle, Washington, USA, 8-12 July 2006. ACM Press.

[414] Edward P. K. Tsang, Jin Li, and James M. Butler. EDDIE beats the bookies. *Software: Practice and Experience*, 28(10):1033–1043, 1998.

[415] A. M. Turing. 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 5. Edinburgh: Edinburgh University Press, 1948.

[416] A. M. Turing. Computing machinery and intelligence. *Mind*, 49:433–460, January 01 1950.

[417] Imran Usman, Asifullah Khan, Rafiullah Chamlawi, and Abdul Majid. Image authenticity and perceptual optimization via genetic algorithm and a dependence neighborhood. *International Journal of Applied Mathematics and Computer Sciences*, 4(1):615–620, 2007.

[418] Seetharaman Vaidyanathan, David I. Broadhurst, Douglas B. Kell, and Royston Goodacre. Explanatory optimization of protein mass spectrometry via genetic search. *Analytical Chemistry*, 75(23):6679–6686, 2003.

[419] Vishwesh Venkatraman, Andrew Rowland Dalby, and Zheng Rong Yang. Evaluation of mutual information and genetic programming for feature selection in QSAR. *Journal of Chemical Information and Modeling*, 44(5):1686–1692, 2004.

[420] Barkley Vowk, Alexander (Sasha) Wait, and Christian Schmidt. An evolutionary approach generates human competitive coreware programs. In Mark Bedau, Phil Husbands, Tim Hutton, Sanjeev Kumar, and Hideaki Sizuki, editors, *Workshop and Tutorial Proceedings Ninth International Conference on the Simulation and Synthesis of Living Systems(Alife XI)*, pages 33–36, Boston, Massachusetts, 12 September 2004. Artificial Chemistry and its applications workshop.

[421] Ivana Vukusic, Sushma Nagaraja Grellscheid, and Thomas Wiehe. Applying genetic programming to the prediction of alternative mRNA splice variants. *Genomics*, 89(4):471–479, April 2007.

[422] Reginald L. Walker. Search engine case study: searching the web using genetic programming and MPI. *Parallel Computing*, 27(1-2):71–89, January 2001.

[423] Paul Walsh and Conor Ryan. Paragen: A novel technique for the autoparallelisation of sequential programs using genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 406–409, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[424] Daniel C. Weaver. Applying data mining techniques to library design, lead generation and lead optimization. *Current Opinion in Chemical Biology*, 8(3):264–270, 2004.

[425] P. A. Whigham. A schema theorem for context-free grammars. In *1995 IEEE Conference on Evolutionary Computation*, volume 1, pages 178–181, Perth, Australia, 29 November - 1 December 1995. IEEE Press.

[426] P. A. Whigham. Search bias, language bias, and genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 230–237, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[427] Darrell Whitley. An overview of evolutionary algorithms: practical issues and common pitfalls. *Information and Software Technology*, 43(14):817–831, 2001.

[428] Darrell Whitley, David Goldberg, Erick Cantu-Paz, Lee Spector, Ian Parmee, and Hans-Georg Beyer, editors. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)*, Las Vegas, Nevada, USA, 10-12 July 2000. Morgan Kaufmann.

[429] L. Darrell Whitley. A Genetic Algorithm Tutorial. *Statistics and Computing*, 4:65–85, 1994.

[430] M. J. Willis, H. G. Hiden, and G. A. Montague. Developing inferential estimation algorithms using genetic programming. In *IFAC/ADCHEM International Symposium on Advanced Control of Chemical Processes*, pages 219–224, Banaff, Canada, 1997.

[431] Mark Willis, Hugo Hiden, Peter Marenbach, Ben McKay, and Gary A. Montague. Genetic programming: An introduction and survey of applications. In Ali Zalzala, editor, *Second International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications, GALESIA*, University of Strathclyde, Glasgow, UK, 1-4 September 1997. Institution of Electrical Engineers.

[432] Garnett Wilson and Malcolm Heywood. Introducing probabilistic adaptive mapping developmental genetic programming with redundant mappings. *Genetic Programming and Evolvable Machines*, 8(2):187–220, June 2007. Special issue on developmental systems.

[433] Man Leung Wong. An adaptive knowledge-acquisition system using generic genetic programming. *Expert Systems with Applications*, 15(1):47–58, 1998.

[434] Man Leung Wong. Evolving recursive programs by using adaptive grammar based genetic programming. *Genetic Programming and Evolvable Machines*, 6(4):421–455, December 2005.

[435] Man Leung Wong and Kwong Sak Leung. Inducing logic programs with genetic algorithms: the genetic logicprogramming system genetic logic programming and applications. *IEEE Expert*, 10(5):68–76, October 1995.

[436] Man Leung Wong and Kwong Sak Leung. Evolving recursive functions for the even-parity problem using genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 11, pages 221–240. MIT Press, Cambridge, MA, USA, 1996.

[437] Man Leung Wong and Kwong Sak Leung. *Data Mining Using Grammar Based Genetic Programming and Applications*, volume 3 of *Genetic Programming*. Kluwer Academic Publishers, January 2000.

[438] Man-Leung Wong, Tien-Tsin Wong, and Ka-Ling Fok. Parallel evolutionary algorithms on graphics processing unit. In David Corne, Zbigniew Michalewicz, Bob McKay, Gusz Eiben, David Fogel, Carlos Fonseca, Garrison Greenwood, Gunther Raidl, Kay Chen Tan, and Ali Zalzala, editors, *Proceedings of the 2005 IEEE Congress on Evolutionary Computation*, volume 3, pages 2286–2293, Edinburgh, Scotland, UK, 2-5 September 2005. IEEE Press.

[439] Andrew M. Woodward, Richard J. Gilbert, and Douglas B. Kell. Genetic programming as an analytical tool for non-linear dielectric spectroscopy. *Bioelectrochemistry and Bioenergetics*, 48(2):389–396, 1999.

[440] Sewall Wright. The roles of mutation, inbreeding, crossbreeding and selection in evolution. In D. F. Jones, editor, *Proceedings of the Sixth International Congress on Genetics*, volume 1, pages 356–366, 1932.

[441] Huayang Xie, Mengjie Zhang, and Peter Andreae. Genetic programming for automatic stress detection in spoken english. In Franz Rothlauf, Jurgen Branke, Stefano Cagnoni, Ernesto Costa, Carlos Cotta, Rolf Drechsler, Evelyne Lutton, Penousal Machado, Jason H. Moore, Juan Romero, George D. Smith, Giovanni Squillero, and Hideyuki Takagi, editors, *Applications of Evolutionary Computing, EvoWorkshops2006: EvoBIO, EvoCOMNET, EvoHOT, EvoIASP, EvoInteraction, EvoMUSART, EvoSTOC*, volume 3907 of *LNCS*, pages 460–471, Budapest, 10-12 April 2006. Springer Verlag.

[442] Masayuki Yangiya. Efficient genetic programming based on binary decision diagrams. In *1995 IEEE Conference on Evolutionary Computation*, volume 1, pages 234–239, Perth, Australia, 29 November - 1 December 1995. IEEE Press.

[443] Xin Yao, Edmund Burke, Jose A. Lozano, Jim Smith, Juan J. Merelo-Guervós, John A. Bullinaria, Jonathan Rowe, Peter Tiňo Ata Kabán, and Hans-Paul Schwefel, editors. *Parallel Problem Solving from Nature - PPSN VIII*, volume 3242 of *LNCS*, Birmingham, UK, 18-22 September 2004. Springer-Verlag.

[444] *Proceedings of the 2000 Congress on Evolutionary Computation CEC00*, La Jolla Marriott Hotel La Jolla, California, USA, 6-9 July 2000. IEEE Press.

[445] Jiangang Yu and Bir Bhanu. Evolutionary feature synthesis for facial expression recognition. *Pattern Recognition Letters*, 27(11):1289–1298, August 2006. Evolutionary Computer Vision and Image Understanding.

[446] Jianjun Yu, Jindan Yu, Arpit A. Almal, Saravana M. Dhanasekaran, Debashis Ghosh, William P. Worzel, and Arul M. Chinnaiyan. Feature selection and molecular classification of cancer using genetic programming. *Neoplasia*, 9(4):292–303, April 2007.

[447] Tina Yu. Hierachical processing for evolving recursive and modular programs using higher order functions and lambda abstractions. *Genetic Programming and Evolvable Machines*, 2(4):345–380, December 2001.

[448] Tina Yu and Shu-Heng Chen. Using genetic programming with lambda abstraction to find technical trading rules. In *Computing in Economics and Finance*, University of Amsterdam, 8-10 July 2004.

[449] Tina Yu, Rick L. Riolo, and Bill Worzel, editors. *Genetic Programming Theory and Practice III*, volume 9 of *Genetic Programming*, Ann Arbor, 12-14 May 2005. Springer.

[450] Byoung-Tak Zhang and Heinz Mühlenbein. Evolving optimal neural networks using genetic algorithms with Occam's razor. *Complex Systems*, 7:199–220, 1993.

[451] Byoung-Tak Zhang and Heinz Mühlenbein. Balancing accuracy and parsimony in genetic programming. *Evolutionary Computation*, 3(1):17–38, 1995.

[452] Byoung-Tak Zhang, Peter Ohm, and Heinz Mühlenbein. Evolutionary induction of sparse neural trees. *Evolutionary Computation*, 5(2):213–236, 1997.

[453] Mengjie Zhang and Will Smart. Using gaussian distribution to construct fitness functions in genetic programming for multiclass object classification. *Pattern Recognition Letters*, 27(11):1266–1274, August 2006. Evolutionary Computer Vision and Image Understanding.

[454] Yang Zhang and Peter I. Rockett. Feature extraction using multi-objective genetic programming. In Yaochu Jin, editor, *Multi-Objective Machine Learning*, volume 16 of *Studies in Computational Intelligence*, chapter 4, pages 79–106. Springer, 2006. Invited chapter.