24 Genetic Programming – Introduction, Applications, Theory and Open Issues

Leonardo Vanneschi[1] · *Riccardo Poli*[2]
[1]Department of Informatics, Systems and Communication, University of Milano-Bicocca, Italy
vanneschi@disco.unimib.it
[2]Department of Computing and Electronic Systems, University of Essex, Colchester, UK
rpoli@essex.ac.uk

Abstract

Genetic programming (GP) is an evolutionary approach that extends genetic algorithms to allow the exploration of the space of computer programs. Like other evolutionary algorithms, GP works by defining a goal in the form of a quality criterion (or fitness) and then using this criterion to evolve a set (or population) of candidate solutions (individuals) by mimicking the basic principles of Darwinian evolution. GP breeds the solutions to problems using an iterative process involving the probabilistic selection of the fittest solutions and their variation by means of a set of genetic operators, usually crossover and mutation. GP has been successfully applied to a number of challenging real-world problem domains. Its operations and behavior are now reasonably well understood thanks to a variety of powerful theoretical results. In this chapter, the main definitions and features of GP are introduced and its typical operations are described. Some of its applications are then surveyed. Some important theoretical results in this field, including some very recent ones, are reviewed and some of the most challenging open issues and directions for future research are discussed.

1 Introduction

Genetic algorithms (GAs) are capable of solving many problems competently. Furthermore, in their simplest realizations, they have undergone a variety of theoretical studies, so that a solid understanding of their properties is now available. Nevertheless, the fixed-length string-type representation of individuals that characterizes GAs is unnatural and overly constraining for a wide set of applications. These include the evolution of computer programs, where, rather obviously, the most natural representation for a solution is a hierarchical, variable size structure rather than a fixed-length character string. In particular, fixed-length strings do not readily support the hierarchical organization of tasks into subtasks typical of computer programs; they do not provide a convenient way of incorporating iteration and recursion; and so on. Above all, traditional GA representations cannot be dynamically varied at run time. The initial choice of string-length limits in advance the number of internal states of the system, thereby setting an upper bound on what the system can learn.

This lack of representation power (already recognized two decades ago, for example, in De Jong (1988)) is overcome by genetic programming (GP), an extension of GAs that largely owes its success to Koza (1992a), who defines GP as:

- a systematic method for getting computers to automatically solve a problem starting from a high-level statement of what needs to be done (Koza and Poli 2003).

In some senses, GP represents an attempt to accomplish one of the most ambitious goals of computer science: being able of specifying what one wants a program to do, but not how, and have the computer figure out an implementation.

At a slightly lower level of abstraction, GP basically works like a GA. As GAs do for fixed length strings of characters, GP genetically breeds a population of computer programs to solve a problem. The major difference between GAs and GP is that in the former the individuals

Comp. by: PBhavani Stage: Proof Chapter ID: 24 Title Name: NaturalComputing
Page Number: 1 Date:10/5/10 Time:15:37:28

in the population are fixed-length strings of characters, in the latter they are hierarchical variable-size structures that represent *computer programs*.

Every programming language (e.g., Pascal or C) is capable of expressing and executing general computer programs. Koza, however, chose the *LISP* (*LIS*t *P*rocessing) language to code his GP implementation. Besides the fact that many sophisticated programming tools were available for LISP at the time, the language presented a variety of advantages. Both programs and data have the same form in LISP, so that it is possible and convenient to treat a computer program as data in the genetic population. This common form of programs and data is the list. Lists can be nested and, therefore, can easily represent hierarchical structures such as *syntax trees*. Syntax trees are particularly suitable as representations of computer programs. Their size, shape, and primitives can be changed dynamically by genetic operators, thereby imposing no *a priori* limit on the complexity of what's evolved by GP. LISP facilitates the programming of such manipulations.

While modern GP implementations are based on C, C++, Java, or Python, rather than LISP and syntax trees are nowadays often represented using a flattened array-based representation, rather than linked lists (typical of LISP), from a logical point of view, programs are still treated and manipulated as trees as Koza did. This form of GP will be called *tree-based GP* hereafter.

While the tree-based representation of individuals is the oldest and most common one, it is not the only form of GP that has been employed to evolve computer programs. In particular, in the last few years, a growing attention has been dedicated by researchers to linear genomes (see for instance Brameier and Banzhaf 2001) and graph-based genomes (see for instance Miller 2001). Many other forms of GP, for example, based on grammars or based on the estimation of probability distributions, have also been proposed. However, for space limitations these forms are not covered in this chapter. The interested reader is referred to Poli et al. (2008) for a more comprehensive review.

This chapter is structured as follows. ➲ Sect. 2 introduces the main definitions and features of GP and its operators. ➲ Sect. 3 offers an outline of some of the most significant real-world applications of GP. ➲ Sect. 4 reviews the most significant theoretical results to date on GP, including some recent and exciting new advances. ➲ Sect. 5 lists some important open issues in GP. ➲ Sect. 6 provides some conclusions.

2 The Mechanics of Tree-Based GP

In synthesis, the GP paradigm works by executing the following steps:

1. Generate an initial population of computer programs (or individuals).
2. Perform the following steps until a termination criterion is satisfied:
   (a) Execute each program in the population and assign it a fitness value according to how well it solves the problem.
   (b) Create a new population by applying the following operations:
      i. Probabilistically select a set of computer programs for mating, on the basis of their fitness (selection).
      ii. Copy some of the selected individuals, without modifying them, into the new population (reproduction).

   iii. Create new computer programs by genetically recombining randomly chosen parts of two selected individuals (crossover).
   iv. Create new computer programs by substituting randomly chosen parts of some selected individuals with new randomly generated ones (mutation).
3. The best computer program which appeared in any generation is designated as the result of the GP process. This result may be a solution (or an approximate solution) to the problem.

   In the following sections, each step of this process is analyzed in detail.

2.1 Representation of GP Individuals

The set of all the possible structures that GP can generate includes all the trees that can be built recursively from a set of function symbols $\mathscr{F} = \{f_1, f_2, \ldots, f_n\}$ (used to label internal tree nodes) and a set of terminal symbols $\mathscr{T} = \{t_1, t_2, \ldots, t_m\}$ (used to label the leaves of GP trees). Each function in the function set $\mathscr{F}$ takes a fixed number of arguments, known as its *arity*. Functions may include arithmetic operations ($+, -, \times$, etc.), mathematical functions (such as `sin`, `cos`, `log`, `exp`), Boolean operations (such as *AND*, *OR*, *NOT*), conditional operations (such as `If-Then-Else`), iterative operations (such as `While-Do`), and other domain-specific functions that may be defined to suit the problem at hand. Each terminal is typically either a variable or a constant.

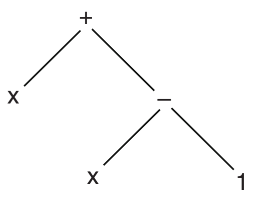
For example, given the set of functions $\mathscr{F} = \{+, -\}$ and the set of terminals $\mathscr{T} = \{x, 1\}$, a legal GP individual is represented in *Fig. 1*. This tree can also be represented by the LISP-like S-expression $(+\ x\ (-\ x\ 1))$ (for a definition of LISP S-expressions see, for instance, Koza (1992)).

It is good practice to choose function and terminal sets that satisfy two requirements: *closure* and *sufficiency.*

The *closure property* requires that each of the functions in the function set be able to accept, as its arguments, any value and data type that may possibly be returned by any function in the function set and any value and data type that may possibly result from the evaluation of any terminal in the terminal set. In other words, each function should be defined (and behave properly) for any combination of arguments that it may encounter. This property is essential since, clearly, programs must be executed in order to assign them a fitness. A failure in the execution of one of the programs composing the population would lead either to a failure of the whole GP system or to unpredictable results.

The function and terminal sets used in the previous example clearly satisfy the closure property. The sets $\mathscr{F} = \{*, /\}$ and $\mathscr{T} = \{x, 0\}$, however, do not satisfy this property. In fact, each evaluation of an expression containing an operation of division by zero would lead to unpredictable behavior. In practical applications, the division operator is often modified in order to make sure the closure property holds.

Fig. 1
A tree that can be built with the sets $\mathscr{F} = \{+, -\}$ and $\mathscr{T} = \{x, 1\}$.



Respecting the closure property in real-world applications is not always straightforward, particularly if the use of different data types is required by the problem domain. A common example is the mixing of Boolean and numeric functions. For example, if one used a function set composed by Boolean functions (*AND*, *OR*, ...), arithmetic functions $(+,-,*,/,\ldots)$, comparison functions $(>,<,=,\ldots)$, and conditionals (*IF THEN ELSE*), expressions such as:

$$IF\ ((x > 10*y)\ AND\ (y > 0))\ THEN\ z + y\ ELSE\ z*x$$

might easily be evolved. In such cases, introducing typed functions and type-respecting genetic operations in the GP system can help enforce closure. This is achieved, for example, by *strongly typed GP* (Banzhaf et al. 1998), a GP system in which each primitive carries information about its type as well as the types it can call, thus forcing functions calling it to cast arguments into the appropriate types. Using types make even more sense in GP than for human programmers, since human programmers have a mental model of what they are doing, whereas the GP system is completely random in its initialization and variation phases. Furthermore, type consistency reduces the search space, which is likely to improve chances of success for the search.

The *sufficiency property* requires that the set of terminals and the set of functions be capable of expressing a solution to the problem. For instance, a function set including the four arithmetic operations combined with a terminal set including variables and constants respects the sufficiency property if the problem at hand requires the evolution of an analytic function (Any function can be approximated to the desired degree by a polynomial via Taylor's expansion.). The primitive set $\mathscr{F} = \{+,-\}$, on the other hand, would not be sufficient, since it can only represent linear functions.

For many domains, there is no way of being sure whether sufficiency holds for the chosen set. In these cases, the definition of appropriate sets depends very much on prior knowledge about the problem and on the experience of the GP designer.

2.2 Initialization of a GP Population

The initialization of the population is the first step of the evolution process. It involves the creation of the program structures that will later be evolved. The most common initialization methods in tree-based GP are the *grow* method, the *full* method and the *ramped half-and-half* method (Koza 1992a). These methods will be explained in the following paragraphs. All are controlled by a parameter: the maximum depth allowed for the trees, d.

When the *grow method* is employed, each tree of the initial population is built using the following algorithm:

- A random symbol is selected with uniform probability from the function set $\mathscr{F}$ to be the root of the tree; let n be the arity of the selected function symbol.
- n nodes are selected with uniform probability from the union of the function set and the terminal set, $\mathscr{F} \cup \mathscr{T}$, to be its children;
- For each function symbol within these n nodes, the grow method is recursively invoked, that is, its children are selected from the set $\mathscr{F} \cup \mathscr{T}$, unless the node has a depth equal to $d-1$. In the latter case, its children are selected from $\mathscr{T}$.

While this method ensures that no parts of the tree are deeper than d, the drawing of primitives from $\mathscr{F} \cup \mathscr{T}$ allows branches to be fully leafed before they reach this maximum depth. So, initial trees of varied shape and size typically result from the use of the grow method. One should note, however, that the exact distribution of tree shapes depends in a nontrivial way on the ratio between the number of primitives of each arity and the number of terminals. If the primitive set contains a large number of terminals, the trees produced by the grow method tend to be small. On the contrary, if there are a few terminals and many functions with an arity of 2 or more, branches will tend to reach the maximum depth d, resulting in almost full trees.

Instead of selecting nodes from $\mathscr{F} \cup \mathscr{T}$, the *full method* chooses only function symbols until the maximum depth is reached. Then it chooses only terminals. The result is that every branch of the tree goes to the full maximum depth.

As first noted by Koza (1992a), population initialized with the above two methods may lack diversity. To fix this, Koza suggested to use a technique he called *ramped half-and-half*. This works as follows. With the ramped half-and-half method, a fraction $\frac{1}{d}$ of the population is initialized with trees having a maximum depth of 1, a fraction $\frac{1}{d}$ with trees of maximum depth 2, and so on. For each depth group, half of the trees are initialized with the full technique and half with the grow technique. In this way, the initial population is composed by a mix of large, small, full, and unbalanced trees, thus ensuring a certain amount of diversity.

2.3 Fitness Evaluation

Each program in the population is assigned a fitness value, representing the degree to which it solves the problem of interest. This value is calculated by means of some well-defined procedure. Two fitness measures are most commonly used in GP: the *raw fitness* and the *standardized fitness*. They are described below.

Raw fitness is the measurement of fitness that is stated in the natural terminology of the problem itself. In other words, raw fitness is the simplest and most natural way to calculate the degree to which a program solves a problem. For example, if the problem consists of controlling a robot so that it picks up as many of the objects contained in a room as possible, then the raw fitness of a program could simply be the number of objects actually picked up at the end of its execution.

Often, but not always, raw fitness is calculated over a set of *fitness cases*. A fitness case corresponds to a representative situation in which the ability of a program to solve a problem can be evaluated. For example, consider the problem of generating an arithmetic expression that approximates the polynomial $x^4 + x^3 + x^2 + x$ over the set of natural numbers smaller than 10. Then, a fitness case is one of those natural numbers. Suppose x^2+1 is the functionality of a program one needs to evaluate. Then, $2^2 + 1 = 5$ is the value taken by this expression over the fitness case 2. Raw fitness is then defined as the sum over all fitness cases of the distances between the target values expected from a perfect solution and the values actually returned by the individual being evaluated. Formally, the raw fitness, f_R, of an individual i, calculated over a set of N fitness cases, can be defined as

$$f_R(i) = \sum_{j=1}^{N} |S(i,j) - C(j)|^k \tag{1}$$

where $S(i,j)$ is the value returned by the evaluation of individual i over fitness case j, $C(j)$ is the correct output value associated to fitness case j and k is some natural number (often either k=1 or k=2).

Fitness cases are typically a small sample of the entire domain space. The choice of how many fitness cases (and which ones) to use is often a crucial one since whether or not an evolved solution will generalize over the entire domain depends on this choice. In practice, the decision is made on the basis of knowledge about the problem and practical performance considerations (the bigger the training set, the longer the time required to evaluate fitnesses). A first theoretical study on the suitable number of fitness cases to be used has been presented in Giacobini et al. (2002).

Standardized fitness is a reformulation of the raw fitness so that lower numerical values are better. When the smaller the raw fitness the better (as in the case in Eq. 1), then the standardized fitness, f_S, can simply be equal to the raw fitness, that is $f_S = f_R$. It is convenient and desirable to ensure the best value of standardized fitness is equal to zero. When this is not the case, this can be obtained by subtracting or adding a constant to the raw fitness. If, for a particular problem, the bigger the raw fitness the better, the standardized fitness f_S of an individual can be defined as $f_S = f_R^{max} - f_R$, where f_R is the raw fitness of the individual and f_R^{max} is the maximum possible value of raw fitness (which is assumed to be known).

2.4 Selection

Each individual belonging to a GP population can undergo three different operations: genetic operators can be applied to that individual, it can be copied into the next generation as it is, or it can be discarded. *Selection* is the operator that decides whether a chance of reproduction is given to an individual. It is a crucial step for GP, as is for all EAs, since the success and pace of evolution often depends on it.

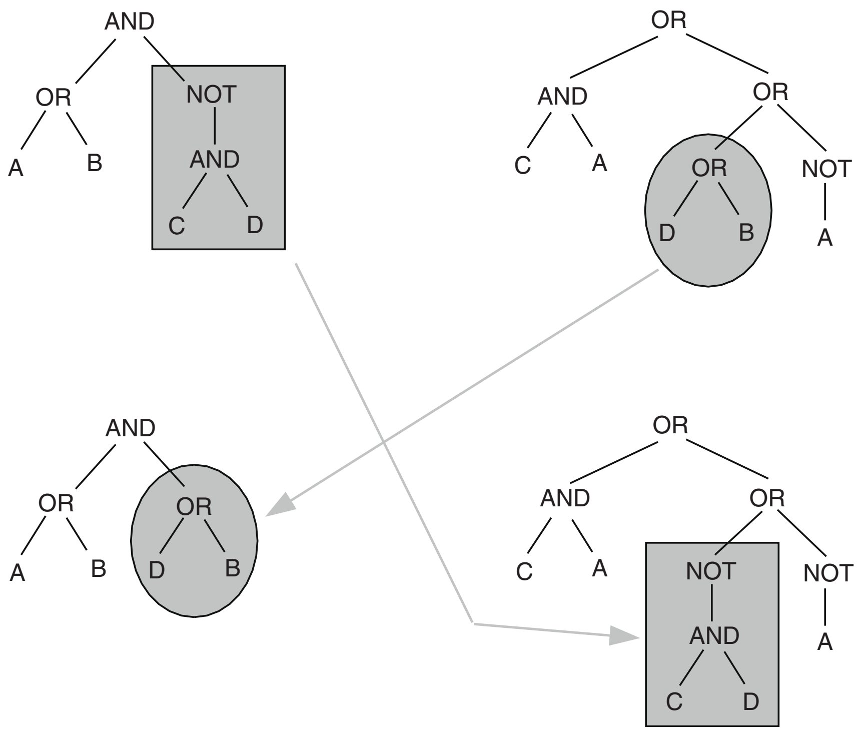
Many selection algorithms have been developed. The most commonly used are *fitness proportionate selection*, *ranking selection*, and *tournament selection*. All of them are based on fitness: individuals with better fitness have higher probability of being chosen for mating. Selection mechanisms used for GP are typically identical to the ones used for GAs and for other EAs. They will not be discussed in detail in this chapter, since they are introduced elsewhere in this book.

2.5 Crossover

The *crossover* or sexual *recombination* operator creates variation in the population by producing offspring that consist of genetic material taken from the parent. The parents, T_1 and T_2, are chosen by means of a selection method.

Standard GP crossover (often called *subtree crossover*) (Koza 1992a) begins by independently selecting a random node in each parent – it will be called the *crossover point* for that parent. Usually crossover points are chosen in such a way that internal nodes are picked with a probability of 0.9 and any node (internal node or terminal) with a probability 0.1. The *crossover fragment* for a particular parent is the subtree rooted at the crossover point. An offspring is produced by deleting the crossover fragment of T_1 from T_1 and inserting the crossover fragment of T_2 at the crossover point of T_1. Some implementations also return a second offspring, which is produced in a symmetric manner. Figure *2* shows an example of standard GP crossover. Because entire subtrees are swapped and because of the closure property of the primitive set, crossover always produces syntactically legal programs.

Fig. 2
An example of standard GP crossover. Crossover fragments are indicated by the *shaded areas*.



It is important to remark that when one of the crossover points is a leaf while the other is the root of a tree the offspring can be much bigger and deeper than the parents. While this may be desirable at early stages of a run, in the presence of the phenomenon of *bloat* – the progressive growth of the code size of individuals in the population generation after generation (more on this in Sect. 4.4) – some limit may have to be imposed on offspring size. Typically, if the offspring's size or depth is beyond a limit, the offspring is either rejected (and crossover attempted again) or accepted in the population but is given a very low fitness.

Many variants of the standard GP crossover have been proposed in literature. The most common ones consist in assigning probabilities of being chosen as crossover points to nodes, based on node depth. In particular, it is very common to assign low probabilities to crossover points near the root and the leaves, so as to reduce the occurrence of the phenomenon described above. Another kind of GP crossover is *one-point crossover*, introduced in Poli and Langdon (1997), where the crossover points in the two parents are forced to be at identical positions. This operator was important it the development of a solid GP theory (see Sect. 4), since it provided a stepping stone that allowed the extension to GP of corresponding GA theory.

2.6 Mutation

Mutation is asexual, that is, it operates on only one parental program.

Standard GP mutation, often called *subtree mutation*, begins by choosing a *mutation point* at random, with uniform probability, within a selected individual. Then, the subtree rooted at the mutation point is removed and a new randomly generated subtree is inserted at that point.

Fig. 3

An example of standard GP mutation.

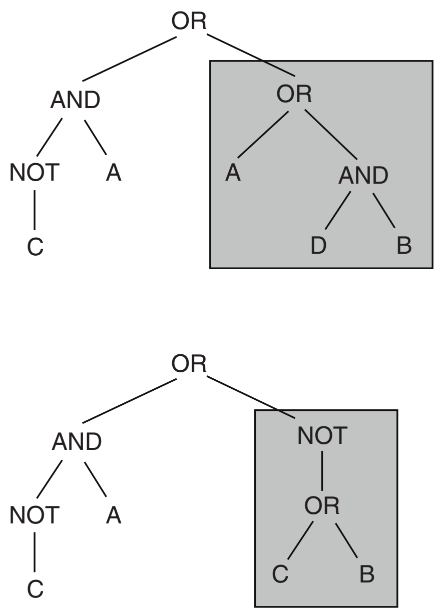


Figure 3 shows an example of standard GP mutation. As it is the case for standard crossover this operation is controlled by a parameter that specifies the maximum depth allowed, thereby limiting the size of the newly created subtree that is to be inserted. Nevertheless, the depth of the generated offspring can be considerably larger than the one of the parent.

Many variants of standard GP mutation have been developed. The most commonly used are the ones aimed at limiting the probability of selecting as mutation points the root and/or the leaves of the parent. Another common form of mutation is *point mutation* (Poli and Langdon 1997) that randomly replaces nodes with random primitives of the same arity. Other commonly used variants of GP mutation are *permutation* (or *swap* mutation) that exchanges two arguments of a node and *shrink mutation* that generates a new individual from a parent's subtree. Another form of mutation is *structural mutation* (Vanneschi 2003, 2004). Being consistent with the edit distance measure, this form of mutation is useful in the study of GP problem difficulty (see Sect. 4.6).

2.7 Other Variants

In GP systems using a *steady state* scheme, the individuals produced by variation operators are immediately inserted into the population without waiting for a full generation to be complete. To keep the population size constant, a corresponding number of individuals need to be removed from the population. The removed individuals are typically the ones with the worst fitness, but versions of GP systems also exist where the removed individuals are randomly selected or where offspring replace their parents. After the new individuals have been inserted into the population, the process is iterated. A GP system using the traditional (non-steady-state) strategy is called *generational*.

Comp. by: PBhavani Stage: Proof Chapter ID: 24 Title Name: NaturalComputing
Page Number: 1 Date:10/5/10 Time:15:37:29

Automatically defined functions (ADFs) are perhaps the best-known mechanisms by which GP can evolve and use subroutines inside individuals. In GP with ADFs, each program is represented by a set of trees, one for each ADF, plus one for the main program. Each ADF may have zero, one, two or more variables that play the role of formal parameters. ADFs allow parametrizes reuse of code and hierarchical invocation of evolved code. The usefulness of ADFs has been shown by Koza (1994).

In the same vein, Koza defined automatically defined iterations (ADIs), automatically defined loops (ADLs), automatically defined recursions (ADRs), and automatically defined stores (ADSs) (Koza et al. 1999). All these mechanisms may allow a high degree of reusability of code inside GP individuals. However, their use is still not very widespread.

2.8 GP Parameters

The user of a GP has to decide the set of functions $\mathcal{F}$ and the set of terminals $\mathcal{T}$ to be used to represent potential solutions of a given problem and the exact implementation of the genetic operators to be used to evolve programs. This is not all, though. The user must also set some parameters that control evolution.

The most important parameters are population size, stopping criterion, technique used to create the initial population, selection algorithm, crossover type and rate, mutation type and rate, maximum tree depth, steady state vs. generational update, presence or absence of ADFs or other modularity-promoting structures, and presence or absence of elitism (i.e., the guaranteed survival of the best individual(s) found in each generation). The setting of these parameters represents, in general, an important choice for the performance of the GP system, although, based on one's experience, one would suggest that population size and anything to do with selection (e.g., elitism, steady state, etc.) are probably the most important ones.

2.9 GP Benchmarks

Koza (1992a) defines a set of problems that can be considered as "typical GP problems," given that they mimic some important real-life applications and they are simple to define and to apply to GP. For this reason, they have been adopted by the GP research community as a, more or less, agreed-upon set of benchmarks, to be used in experimental studies. They include the following:

- *The Even k Parity Problem*, whose goal is to find a Boolean function of k arguments that returns *true* if an even number of its arguments evaluates to true, and *false* otherwise.
- *The h Multiplexer Problem*, where the goal is to design a Boolean function with h inputs that approximates a multiplexer function.
- *The Symbolic Regression Problem*, that aims at finding a program that matches a given target mathematical function.
- *The Intertwined Spirals problem*, the goal of which is to find a program to classify points in the $x-y$ plane as belonging to one of two spirals.
- *The Artificial Ant on the Santa Fe trail*, whose goal is to find a target navigation strategy for an agent on a limited toroidal grid.

Comp. by: PBhavani Stage: Proof Chapter ID: 24 Title Name: NaturalComputing
Page Number: 1 Date:10/5/10 Time:15:37:29

3 Examples of Real-World Applications of GP

Since its early beginnings, GP has produced a cornucopia of results. Based on the experience of numerous researchers over many years, it appears that GP and other evolutionary computation methods have been especially productive in areas having some or all of the following properties:

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

The literature, which covers more than 5,000 recorded uses of GP, reports an enormous number of applications where GP has been successfully used as an automatic programming tool, a machine learning tool, or an automatic problem-solving engine. It is impossible to list all such applications here. Thus, to give readers an idea of the variety of successful GP applications, below a representative sample is listed (see Poli et al. (2008) for a more detailed analysis):

- *Curve fitting, data modelling, and symbolic regression.* A large number of GP applications are in these three fields (see, e.g., Lew et al. 2006 and Keijzer 2004). GP can be used as a stand-alone tool or coupled with one of the numerous existing feature selection methods, or even GP itself can be used to do feature selection (Langdon and Buxton 2004). Furthermore, GP can also be used in cases where more than one output (prediction) is required. In that case, linear GP with multiple output registers, graph-based GP with multiple output nodes, or a GP operating on single trees with primitives on vectors can be used.
- *Image and Signal Processing.* The use of GP to process surveillance data for civilian purposes, such as predicting motorway traffic jams from subsurface traffic speed measurements was suggested in Howard and Roberts (2004); GP found recurrent filters in Esparcia–Alcazar and Sharman (1996); the use of GP to preprocess images, particularly of human faces, to find regions of interest for subsequent analysis was presented in Trujillo and Olague (2006). Classifications of objects and human speech with GP was achieved in Zhang and Smart (2006) and Xie et al. (2006), respectively. A GP system in which the image processing task is split across a swarm of evolving agents and some of its applications have been described in Louchet (2001). Successful GP applications in medical imaging can be found, for instance, in Poli (1996).
- *Financial Trading, Time Series, and Economic Modelling.* Recent papers have looked at the modeling of agents in stock markets (Chen and Liao 2005), evolving trading rules for the S&P 500 (Yu and Chen 2004), and forecasting the Heng-Sheng index (Chen et al. 1999). Other examples of financial applications of GP include, for example, Jin and Tsang (2006) and Tsang and Jin (2006).

- *Industrial Process Control.* GP is frequently used in industrial process control, although, of course, most industrialists have little time to spend on academic reporting. A notable exception is Dow Chemical, where a group has been very active in publishing results (Castillo et al. 2006; Jordaan et al. 2006). Reference [54] describes where industrial GP stands now and how it will progress. Other interesting contributions in related areas are, for instance, Dassau et al. (2006) and Lewin et al. (2006). GP has also been applied in the electrical power industry (Alves da Silva and Abrao 2002).
- *Medicine, Biology, and Bioinformatics.* Among other applications, GP is used in biomedical data mining. Of particular medical interest are very wide data sets, with typically few samples and many inputs per sample. Recent examples include single nuclear polymorphisms (Shah and Kusiak 2004), chest pain (Bojarczuk et al. 2000), and Affymetrix GeneChip microarray data (Langdon and Buxton 2004; Yu et al. 2007).
- *Computational Chemistry.* Computational chemistry is important in the drug industry. However, the interactions between chemicals that might be used as drugs are beyond exact calculation. Therefore, there is great interest in approximate models that attempt to predict either favorable or adverse interactions between proto-drugs and biochemical molecules. These models can be applied very cheaply in advance of the manufacturing of chemicals, to decide which of the myriad of chemicals is worth further study. Examples of GP approaches include Barrett and Langdon (2006); Hasan et al. (2006), and Archetti et al. (2007).
- *Entertainment, Computer Games, and Art.* Work on GP in games includes, among others, (Azaria and Sipper 2005; Langdon and Poli 2005). The use of GP in computer art is reported for instance in Jacob (2000, 2001). Many recent techniques are described in Machado and Romero (2008).
- *Compression.* GP has been used to perform lossy compression of images (Koza 1992b; Nordin and Banzhaf 1996) and to identify iterated functions system, which are used in the domain of fractal compression (Lutton et al. 1995). GP was also used to evolve wavelet compression algorithms (Klappenecker and May 1995) and nonlinear predictors for images (Fukunaga and Stechert 1998). Recently, a GP system called *GP-ZIP* has been proposed that can do for lossless data compression (Kattan and Poli 2008).
- *Human-competitive results.* Getting machines to produce competitive results is the very reason for the existence of the fields of AI and machine learning. Koza et al. (1999) proposed that an automatically created result should be considered "human-competitive" if it satisfies at least one of eight precise criteria. These include patentability, beating humans in regulated competitions, producing publishable results, etc. Over the years, dozens of GP results have passed the human-competitiveness test (see, Trujillo and Olague 2006 and Hauptman and Sipper 2007).

4 GP Theory

As discussed in ➲ Sect. 3, GP has been remarkably successful as a problem-solving and engineering tool. One might wonder how this is possible, given that GP is a nondeterministic algorithm, and, as a result, its behavior varies from run to run. Why can GP solve problems and how? What goes wrong when it cannot? What are the reasons for certain undesirable behaviors, such as bloat? Below a summary of current research will be given.

4.1 Schema Theories

If one could visualize the search performed by GP, one 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, one would observe different trajectories. If one could see regularities, these might provide one with a deep understanding of how the algorithm is searching the program space for the solutions, and perhaps help one see why GP is successful in finding solutions in certain runs and unsuccessful in others. Unfortunately, it is normally impossible to exactly visualize the program search space due to its high dimensionality and complexity, making this approach nonviable.

An alternative approach to better understanding the dynamics of GP is to study mathematical models of evolutionary search. Exact mathematical models of GP are probabilistic descriptions of the operations of selection, reproduction, crossover, and mutation. They explicitly represent how these operations determine which areas of the program space will be sampled by GP, and with what probability. There are a number of cases where this approach has been very successful in illuminating some of the fundamental processes and biases in GP systems.

Schema theories are among the oldest and the best-known models of evolutionary algorithms (Holland 1975). These 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 (Stephens and Waelbroeck 1999) provides exact information about the distribution of the population at the next generation in terms of quantities measured at the current generation, without having to actually run the algorithm.

The theory of schemata in GP has had a difficult childhood. However, recently, the first exact GP schema theories have become available (Poli 2001), which give exact formulations for the expected number of individuals sampling a schema at the next generation. Initially, these exact theories were only applicable to GP with one-point crossover (see Sect. 2.5). However, more recently, they have been extended to other types of crossover including most crossovers that swap subtrees (Poli and McPhee 2003a,b).

Other models of evolutionary algorithms include models based on Markov chain theory. These models are discussed in the next section.

4.2 Markov Chains

Markov chains are important in the theoretical analysis of evolutionary algorithms operating on discrete search spaces (e.g., Davis and Principe 1993). Since such algorithms are stochastic but Markovian (e.g., in GAs and GP the population at the next generation typically depends only on the population at the current generation), all one needs to do to study their behavior is to compute with which probability an algorithm in any particular state (e.g., with a particular population composition) will move any other state (e.g., another population composition) at the next generation. These probabilities are stored in a stochastic matrix M, which therefore describes with which probability the algorithm will exhibit all possible behaviors over one generation. By taking powers of the matrix M and multiplying it by the initial probability distribution over states determined by the initialization algorithm, it is then possible to study the behavior of the algorithm over any number of generations. It is, for example, possible to compute the probability of solving a problem within *n* generations, the first hitting time, the average and best fitness after *n* generations, etc.

Markov models have been applied to GP (Poli and McPhee 2003; Mitanskiy and Rowe 2006), but so far they have not been developed as fully as the schema theory model.

4.3 Search Spaces

The models presented in the previous two sections treat the fitness function as a black box. That is, there is no representation of the fact that in GP, unlike in other evolutionary techniques, the fitness function involves the execution of computer programs on a variety of inputs. In other words, schema theories and Markov chains do not tell how fitness is distributed in the search space.

The theoretical characterization of the space of computer programs explored by GP from the point of view of fitness distributions aims at answering this very question (Langdon and Poli 2002). The main result in this area is the discovery 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. This happens in a variety of systems and can be proven mathematically for LISP expressions (without side effects) and machine code programs without loops (e.g., see Langdon and Poli 2002). Recently, Poli and Langdon (2006) started extending these results to Turing complete machine code programs. A mathematical analysis of the halting process based on a Markov chain model of program execution and halting was performed, which derived a scaling law for the *halting probability* for programs as their length is varied.

4.4 Bloat

Very often, the average size (number of nodes) of the programs in a GP population, after a certain number of generations in which it is largely static, starts growing at a rapid pace. Typically, the increase in program size is not accompanied by any corresponding increase in fitness. This phenomenon is known as *bloat*. Its origin has been the focus of intense research for over a decade. Bloat is not only interesting from a scientific point of view: it also has significant practical deleterious effects, slowing down the evaluation of individuals, and often consuming large computational resources. There are several theories of bloat.

For instance, the *replication accuracy theory* (McPhee and Miller 1995) states that the success of a GP individual depends on its ability to have offspring that are functionally similar to the parent: as a consequence, GP evolves toward (bloated) representations that increase replication accuracy.

The nodes in a GP tree can be categorized into two classes: *inactive code* (code that is not executed or has no effect) and *active code* (all code that is not inactive). The *removal bias theory* (Soule and Foster 1998b) observes that inactive code in a GP tree tends to be low in the tree, residing in 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 while retaining the fitness of their parent, leading ultimately to growth in the average program size.

The *nature of program search spaces theory* (Langdon et al. 1999) relies on the result mentioned above that 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.

In Poli and Mcphee (2003), a *size evolution equation* for GP was developed, which provided an exact formalization of the dynamics of average program size. The original equation was derived from the exact schema theory for GP. This equation can be rewritten in terms of the expected change in average program size as

$$E[\Delta\mu(t)] = \sum_{\ell} \ell \times (p(\ell,t) - \Phi(\ell,t)) \tag{2}$$

where $\Phi(\ell,t)$ is the proportion of programs of size ℓ in the current generation and $p(\ell,t)$ is their selection probability. This equation does not directly explain bloat, but it constrains what can and cannot happen size-wise in GP populations. So, any explanation for bloat (including the theories above) has to agree with it.

The newest, and hopefully conclusive, explanation for bloat has been recently formalized in the so-called *crossover bias theory* (Dignum and Poli 2007; Poli et al. 2007), which is based on Eq. 2. On average, each application of subtree crossover removes as much genetic material as it inserts; consequently crossover on its own does not produce growth or shrinkage. While the *mean* program size is unaffected, however, *higher moments* of the distribution are. In particular, crossover pushes the population toward a particular distribution of program sizes, known as a *Lagrange distribution of the second kind*, where small programs have a much higher frequency than longer ones. For example, crossover generates a very high proportion of single-node individuals. In virtually all problems of practical interest, however, very small programs have no chance of solving the problem. As a result, programs of above average size have a selective advantage over programs of below average size, and the mean program size increases. Because crossover will continue to create small programs, the increase in average size will continue generation by generation.

Numerous empirical techniques have been proposed to control bloat (Soule and Foster 1998; Langdon et al. 1999). These include the use of size and depth limits, the use of genetic operators with an inbuilt anti-bloat bias, the use of multi-objective selection techniques to jointly optimize fitness and minimize program size. Among these are the famous parsimony pressure method (Koza 1992b; Zhang and Mühlenbein 1995) and a recent improvement of if, the covariant parsimony pressure method (Poli and McPhee 2008). The reader is referred to Poli et al. (2008), Chap. 11] for a survey.

4.5 Is GP Guaranteed to Find Solutions to Problems?

While often Markov transition matrices describing evolutionary algorithms are very large and difficult to manage numerically, it is sometimes possible to establish certain properties of such matrices theoretically. An important property in relation to the ability of an algorithm to reach all possible solutions to a problem (given enough time) is the property of *ergodicity*. What it means in practice is that there exist some $k \in \mathbb{N}$ such that all elements of M^k are nonzero. In this case, $\lim_{k\to\infty} M^k$ is a particular matrix where all elements are nonzero. This means that irrespective of the starting state, the probability of reaching a solution to a problem (e.g., the global optimum of a function, if one is using EAs for function optimization) is nonzero. As a result, given enough generations, the algorithm is guaranteed to find a solution.

In the context of EAs acting on traditional fixed-length representations, this has led to proving that EAs with nonzero mutation rates are global optimizers with guaranteed convergence (Rudolph 1994). While these results have been extended to more general search spaces (Rudolph 1996) and, as it is discussed in Sect. 4.2, Markov chain models of some forms of GP have been recently derived (e.g., see Poli et al. (2004)), the calculations involved in setting up and studying these models are of considerable mathematical complexity. Furthermore, extending the work to infinite search spaces (which is required to model the traditional form of GP where operators can create trees of potentially any size over a number of generations) presents even bigger challenges. So, it is fair to say that at the present time there is no formal mathematical proof that a GP system can always find solutions.

Despite these formal obstacles, it is actually surprisingly easy to find good reasons in support of the conjecture that, with minor modifications, the traditional form of GP is guaranteed to visit a solution to a problem, given enough generations. In fact, if one uses mutation and the mutation operator is such that, albeit with a very low probability, any point in the search space can be generated by mutating any other point, then it is clear that sooner or later the algorithm will visit a solution to the problem. It is then clear that in order to ensure that GP has a guaranteed asymptotic ability to solve all problems, all one needs to do is to add to the mix of operators already present in a GP system some form of mutation that has a nonzero probability of generating any tree in the search space. One such mutation operator is the subtree mutation operator described in Sect. 2.6.

4.6 Problem Difficulty in GP

What problems are solvable via GP? Is it possible to measure the difficulty of a problem for GP? In classical algorithms, one can have a well-developed complexity theory that allows one to classify problems into complexity classes such that problems in the same class have roughly the same complexity, that is, they consume (asymptotically) the same amount of computational resources, usually time. Although, properly speaking, GP is a randomized heuristic and not an algorithm, it would be nice if one were able to somehow classify problems according to some measure of *difficulty.*

Difficulty studies have been pioneered in the related but simpler field of GAs, by Goldberg and coworkers (e.g., see Goldberg 1989 and Horn and Goldberg 1995). Their approach consisted in constructing functions that should *a priori* be easy or hard for GAs to solve. These approach has been followed by many others (e.g., Mitchell et al. 1992 and Forrest and Mitchell 1993) and have been at least partly successful in the sense that they have been the source of many ideas as to what makes a problem easy or difficult for GAs. One concept that underlies many measures of difficulty is the notion of *fitness landscapes.*

The metaphor of a fitness landscape (Stadler 2002), although not without faults, has been a fruitful one in several fields and is attractive due to its intuitiveness and simplicity. Probably, the simplest definition of fitness landscape in EAs is a plot where the points in the abscissas represent the different individual genotypes in a search space and the ordinates represent the fitness of each of these individuals (Langdon and Poli 2002). If genotypes can be visualized in two dimensions, the plot can be seen as a three-dimensional "map," which may contain peaks and valleys. The task of finding the best solution to the problem is equivalent to finding the highest peak (for maximization problems).

One early example of the application of the concept of fitness landscapes to GP is represented by the work of Kinnear (1994), where GP difficulty was related to the shape of the fitness landscape and analyzed through the use of the fitness *auto-correlation function*. Langdon and Poli (2002) took an experimental view of GP fitness landscapes in several works summarized in their book. After selecting important and typical classes of GP problems, they study fitness landscapes either exhaustively, whenever possible, or by randomly sampling the program space when enumeration becomes unfeasible. Their work highlights several important characteristics of GP spaces, such as density and size of solutions and their distribution. This is useful work. Even if the authors' goals were not establishing problem difficulty, it certainly has a bearing on it. More recently, Langdon (2003) has extended his studies for convergence rates in GP for simple machine models (which are amenable to quantitative analysis by Markov chain techniques) to convergence of program fitness landscapes for the same machine models using genetic operators and search strategies to traverse the space. This approach is rigorous because the models are simple enough to be mathematically treatable. The ideas are thus welcome, although their extension to standard GP might prove difficult.

The work of Nikolaev and Slavov (1998) represents the first effort to apply a difficulty measure called *fitness distance correlation* (FDC) (first studied by Jones (1995) for GAs) to GP, with the aim of determining which mutation operator, among a few that they propose, "sees" a smoother landscape on a given problem. Pros and cons of FDC as a general difficulty measure in GP were investigated in Vanneschi (2004), and Tomassini et al. (2005). Those contributions claimed the reliability of FDC and pointed out, as one of its major drawbacks, its lack of predictiveness of FDC (the genotype of the global optima must be known beforehand to calculate it).

In order to overcome this limitation, a new hardness measure, called *negative slope coefficient* (NSC) has been recently presented and its usefulness for GP investigated in Vanneschi et al. (2004, 2006). Results indicated that, although NSC may still need improving and is not without faults, it is a suitable hardness indicator for many well-known GP benchmarks, for some hand-tailored theoretical GP problems and also for some real-life applications (Vanneschi 2007).

Even though interesting, these results still fail to take into account many typical characteristics of GP. For instance, the NSC definition only includes (various kinds of) mutation, but it does not take crossover into account. A similarity/dissimilarity measure related to standard subtree crossover has been presented in Gustafson and Vanneschi (2005, 2008) and Vanneschi et al. (2006). This could be used in the future to build new measures of problem difficulty.

4.7 Does No-Free-Lunch Apply to GP?

Informally speaking, the no-free-lunch (NFL) theory originally proposed by Wolpert and Macready (1997) states that, when evaluated over all possible problems, all algorithms are equally good or bad irrespective of our evaluation criteria.

In the last decade-there have been a variety of results which have refined and specialized NFL (see Whitley and Watson 2005). One such result states that if one selects a set of fitness functions that are *closed under permutation*, the expected performance of any search algorithm over that set of problems is constant, that is, it does not depend on the algorithm one chooses (Schumacher et al. 2001). What does it mean for a set of functions to be closed under permutation? A fitness function is an assignment of fitness values to the elements of the search space. A permutation of a fitness function is simply a rearrangement or reshuffling of the fitness values originally allocated to the objects in the search space.

A set of problems/fitness functions is closed under permutation, if, for every function in the set, all possible shuffles of that function are also in the set.

The result is valid for any performance measure. Furthermore, Schumacher et al. (2001) showed that it is also the case that two arbitrary algorithms will have identical performance over a set of functions only if that set of functions is closed under permutation.

Among the many extension of NFL to a variety of domains, Woodward and Neil (2003) have made some progress in assessing the applicability of NFL to the search spaces explored by GP. In particular, they argued that there is a free lunch in a search space whenever there is a nonuniform many-to-one genotype–phenotype mapping, and that the mapping from syntax to functionality in GP is one such mapping. The reason why NFL would not normally be applicable to search in program spaces is that there are many more programs than functionalities and that not all functionalities are equally likely. So, interpreting syntax trees as genotypes and functionalities as phenotypes, the GP genotype–phenotype mapping is many-to-one and nonuniform, which invalidates NFL.

Beyond this interesting counterexample, to show that in general not all functionalities are equally likely in program search spaces, Woodward and Neil (2003) referred to the results on the limiting distribution of functionality reviewed above and to the universal distribution (Kirchherr et al. 1997) (informally this states that there are many more programs with a simple functionality than programs with a complex one). The latter result, however, applies to Turing complete languages, that is, to programs with memory and loops. So, Woodward and Neil proof applies only to Turing-complete GP, which is essentially a rarity.

In very recent work (Poli et al. 2009) it has been possible to extend these results to the case of standard GP problems and systems and to ascertain that a many-to-one genotype–phenotype mapping is not the only condition under which NFL breaks down when searching program spaces. The theory is particularly easy to understand because it is based on geometry. It is briefly reviewed in the rest of this section.

As seen above, the fitness of a program p in GP is often the result of evaluating the behavior of p in a number of fitness cases and adding up the results (Poli et al. 2008). That is

$$f(p) = \sum_{i=1}^{n} g(p(x_i), t(x_i)) \tag{3}$$

where f is the fitness function, $\{x_i\}$ is a set of fitness cases of cardinality n, and g is a function that evaluates the degree of similarity of its arguments. Almost invariably, the function g respects the axioms of a metric, the most common form of g being $g(a, b) = |a - b|^k$ for $k=1$ or $k=2$.

One can consider a finite space of programs $\Omega = \{p_i\}_{i=1}^{r}$, such as, for example, the space of all possible programs with at most a certain size or depth. A fitness function f over Ω can be represented as a vector $\mathbf{f} = (f_1, \ldots, f_r)$ where $f_i = f(p_i)$. Using this vector representation for fitness functions, one can write Eq. 3 as

$$\mathbf{f} = \left(\sum_{i=1}^{n} g(p_1(x_i), t(x_i)), \ldots, \sum_{i=1}^{n} g(p_r(x_i), t(x_i)) \right) \tag{4}$$

As one can see from Eq. 4, if n and the set of fitness cases $\{x_i\}$ are fixed a priori, then a fitness function is fully determined by the value of the vector of target behaviors, $\mathbf{t} = (t_1, t_2, \ldots, t_n)$, where $t_i = t(x_i)$ is fixed. If one focuses on the most frequent type of program induction application in GP, symbolic regression, one can typically have that the components t_i are scalars representing the desired output for each fitness case.

The function g is a distance (on $\mathbb{R}$). Because the sum of distances is also a distance, one can define the following distance function:

$$d(\mathbf{u}, \mathbf{v}) = \sum_{i=1}^{n} g(\mathbf{u}_i, \mathbf{v}_i) \tag{5}$$

where $\mathbf{u}, \mathbf{v} \in \mathbb{R}^n$. With this in hand, one can see that the fitness associated to a program p can be interpreted as the distance between the vector $\mathbf{t}$ and the vector $\mathbf{p} = (p(x_1), p(x_2), \ldots, p(x_n))$. That is

$$f(\mathbf{p}) = d(\mathbf{p}, \mathbf{t}) \tag{6}$$

Note that whenever one represents programs using their behavior vector one is essentially focusing on the phenotype-to-fitness mapping, thereby complementing the analysis of Woodward and Neil (2003) summarized above.

Using distances, Eq. 4 can be rewritten more concisely as

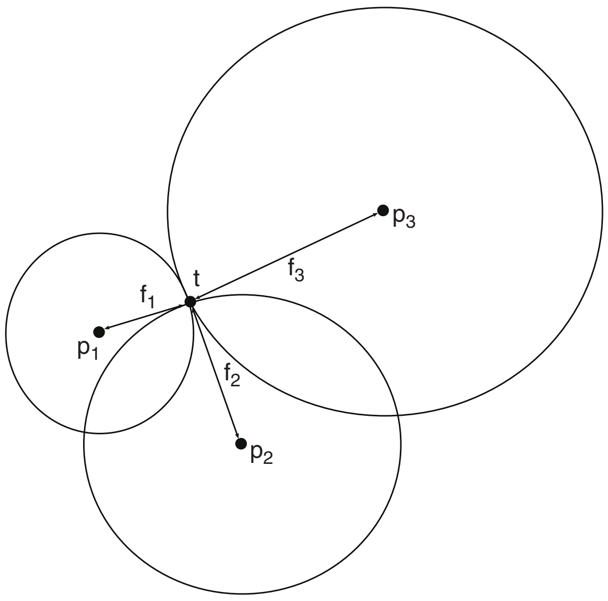
$$\mathbf{f} = (d(\mathbf{p}_1, \mathbf{t}), d(\mathbf{p}_2, \mathbf{t}), \ldots, d(\mathbf{p}_r, \mathbf{t})) \tag{7}$$

Note that if one knows the fitness f of a program p, one knows that the target behavior $\mathbf{t}$ that generated that fitness must be on the surface of a sphere centered on $\mathbf{p}$ (the vector representation of p) and of radius f. So, for every valid symbolic regression fitness function, the target behavior is at the intersection of the spheres centered on the behavior of each program in the search space (see *Fig. 4*).

With this geometric representation of symbolic regression problems in hand, it is not difficult to find out under which conditions NFL does or does not hold for GP. In particular, Poli et al. (2009) showed that:

Fig. 4

If *g* measures the squared difference between two numbers, a valid fitness function requires the spheres centered on each program behavior and with radius given by their corresponding fitness to meet in one point: the target vector t.



Theorem 1 (NFL for GP). *Let* $\mathscr{F} = \{\mathbf{f}_1, \mathbf{f}_2, \ldots, \mathbf{f}_m\}$ *be a set of fitness functions of the form in* Eq. 7 *and let* $\mathscr{T} = \{\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_m\}$ *be the set of target vectors associated to the functions in* $\mathscr{F}$, *with* $\mathbf{t}_i$ *being the vector generating* $\mathbf{f}_i$ **for all i. The set** $\mathscr{F}$ *is closed under permutation (and NFL applies to it)* if and only if *for all target vectors* $\mathbf{t} \in \mathscr{T}$ *and for all permutations* σ *of* $(1, 2, \ldots, r)$ *there exists a target vector* $\tilde{\mathbf{t}} \in \mathscr{T}$ *such that:*

$$\sum_{i=1}^{n} g(p_{\sigma(j)}(x_i), t_i) = \sum_{i=1}^{n} g(p_j(x_i), \tilde{t}_i) \tag{8}$$

or, equivalently,

$$d(\mathbf{p}_{\sigma(j)}, \mathbf{t}) = d(\mathbf{p}_j, \tilde{\mathbf{t}}) \tag{9}$$

for all $j = 1, 2, \ldots, r$.

Equation 9 is a mathematical statement of the geometric requirements for $\tilde{\mathbf{t}}$ to exist. Namely, the target vector $\mathbf{t} \in \mathscr{T}$ associated to a function $\mathbf{f} \in \mathscr{F}$ must be at the intersection of the spheres centered on programs $\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_r$ and having radii $f_1, f_2, \ldots, f_r$, respectively. Permuting the elements of **f** via a permutation σ to obtain a new fitness function corresponds to shuffling the radii of the spheres centered on $\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_r$. Note these centers remain fixed since they represent the behavior of the programs in the search space. After the shuffling, some spheres may have had their radius decreased, some increased, and some unchanged. If any radius has changed, then **t** can no longer be the intersection of the spheres. However, one must be able to find a new intersection $\tilde{\mathbf{t}} \in \mathscr{T}$ or else the new fitness function one has generated is not a symbolic regression fitness function and therefore cannot be a member of $\mathscr{F}$, which would imply that the set is not closed under permutation.

One should also note that Theorem 1 focuses on the set of program behaviors. So, it tells something about the nature of the phenotype-to-fitness function. It really states under which conditions there can be an NFL for a searcher exploring program behaviors with no resampling. If a search algorithm instead explores the space of syntactic structures representing programs, in the presence of symmetries (such as those highlighted by Woodward and Neil), then the searcher will produce resampling of program behaviors even if it never resampled the same syntactic structure. So, in the presence of a set of behaviors for which NFL holds, this would unavoidably give the "syntactic" searcher lower average performance than an algorithm that never resampled behaviors.

This theorem is useful since, by designing fitness functions that break its assumptions, one can find conditions where there is a free lunch for GP. It turns out that such conditions are really easy to satisfy. In particular, it is easy to see that if two programs have identical behavior, they must have identical fitness or there cannot be any intersection between the spheres centered around them. Formally:

Theorem 2 *Consider a search space which includes at least two programs* p_1 *and* p_2 *such that* $p_1(x) = p_2(x)$ *for all x in the training set. Let a set of symbolic regression problems* $\mathscr{F}$ *contain a fitness function f induced by a target vector* **t** *such that there exists a third program* p_3 *in the search space with fitness* $f(p_3) \neq f(p_1) = f(p_2)$. *Then* $\mathscr{F}$ *cannot be closed under permutation and NFL does not hold.*

Note that this result is valid for search in the space of syntactic structures representing programs. As one can see, a many-to-one mapping is not required for a free lunch to be available.

Continuing with the geometric investigation of NFL, it is clear that a necessary condition for the existence of a target vector **t**, which induces a particular fitness function, is that the triangular inequality be verified. More precisely:

Lemma 1 *If a target vector* **t** *induces a symbolic regression fitness function* **f**, *then for every pair of program behaviors* $\mathbf{p}_1$ *and* $\mathbf{p}_2$ *the distance* $d(\mathbf{p}_1,\mathbf{p}_2)$ *between* $\mathbf{p}_1$ *and* $\mathbf{p}_2$ *(based on the same metric used to measure fitnesses) must not be greater than* $f_1 + f_2$.

From this result, one can see that another common situation where there is incompatibility between an assignment of fitness to programs and the fitness representing a symbolic regression problem is the case in which two programs have different behaviors (and so are represented by two distinct points in the $\mathbb{R}^n$), but the sum of their fitnesses is smaller than their distance. This leads to the following result:

Theorem 3 *Given a set of symbolic regression functions* $\mathcal{F}$, *if there exists any function* $\mathbf{f} \in \mathcal{F}$ *and any four program behaviors* $\mathbf{p}_1,\mathbf{p}_2,\mathbf{p}_3,\mathbf{p}_4$ **in a search space such that** $\mathbf{d}(\mathbf{p}_1,\mathbf{p}_2) > f_3 + f_4$ *then the set is not closed under permutation and NFL does not apply.*

So, it is extremely easy to find realistic situations in which a set of symbolic regression problems is provably not closed under permutation. This implies that, in general, one can expect that there is a free lunch for techniques that sample the space of computer programs for the purpose of fitting data sets. This is particularly important since it implies that, for GP, unlike other areas of natural computing, it is worthwhile to try to come up with new and more powerful algorithms.

5 Open Issues

Despite the successful application of GP to a number of challenging real-world problem domains and the recent theoretical advancements in explaining the behavior and dynamics of GP, there remains a number of significant open issues. These must be addressed for GP to become an even more effective and reliable problem-solving method.

The following are some of the important open issues:

- Despite some preliminary studies trying to remove this limit, at present, GP is fundamentally a "static" process, in the sense that many of its characteristics are fixed once and for all before executing it. For instance, the population size, the language used to code individuals ($\mathcal{F}$ and $\mathcal{T}$ sets), the genetic operators used to explore the search space and the fitness function(s) used are all static.
- Directly related to the previous point, until now GP has been mainly used for static optimization problems. No solid explorations, neither theoretical nor applicative, exist of dynamic optimization problems, where for instance the target function (and thus the fitness value of the candidate solutions) change over time.
- While a large body of literature and well-established results exist concerning the issue of generalization for many non-evolutionary machine-learning strategies, this issue in GP has not received the attention it deserves. Only recently, a few papers dealing with the problem of generalization have appeared. Nonetheless, generalization is one of the most important performance evaluation criteria for artificial learning systems and its foundations for GP still have to be deeply understood and applied.

- Although some constructs for handling modularity in GP have been defined (some of them, like ADFs, have been discussed in Sect. 2.7) and widely used, still some questions remain unanswered. Can these constructs help in solving substantially more complex problems? Can one achieve modularity, hierarchy, and reuse in a more controlled and principled manner? Can one provide insights and metrics on how to achieve this?

In the rest of this section, some of these grand challenges and open issues will be expanded upon. It is hoped that this chapter will help focus future research in order to deepen understanding of the method to allow the development of more powerful problem-solving algorithms.

5.1 Dynamic and/or Self-Adapting GP Frameworks

Although the difficulty measures discussed in Sect. 4.6 are useful to statically calculate the hardness of a given problem for GP starting from its high-level specifications, they have not been applied to dynamically improve the search as yet. Indeed, a dynamic modification "on the fly" (i.e., during the evolutionary process) of the fitness function, the population size, the genetic operators, or representation used, driven by one or more hardness measures represents one of the most challenging open issues in GP, as in many other fields of evolutionary computation. One of the major difficulties in the realization of such a dynamic and self-adapting GP environment is probably represented by the fact that both FDC and NSC have always been calculated, until now, using a large sample of the search space and not the evolving population that is typically much smaller. Furthermore, the lack of diversity in populations after some generations may seriously compromise the reliability of these indicators. Nevertheless, this issue remains a challenge and a first attempt in this direction has recently been presented in Wedge and Kell (2008).

Models using dynamically sized populations, although not based on the values of difficulty indicators, have been presented, among others, in Rochat et al. (2005), Tomassini et al. (2004), and Vanneselni (2004), where the authors show that fixed-size populations may cause some difficulties to GP. They try to overcome these difficulties by removing or adding new individuals to the population in a dynamic way during evolution. In particular, individuals are removed as long as the best or average population's fitness keeps improving, while new individuals (either randomly generated ones or mutations of the best ones in the population) are added when fitness stagnates. The authors show a set of experimental results that confirm the benefits of the presented methods, at least for the studied problems. These results suggest that fixed-size population unnaturally limits the GP behavior and that, according to what happens in nature, the evolution has a benefit when populations are left free to shrink or increase in size dynamically, according to a set of events. Some of the most relevant and macroscopic of these events are so called *plagues* that usually cause a violent decrease in natural population sizes. Plagues have been successfully applied to GP in Fernandez et al. (2003a,b) and Fernandez and Martin (2004). These contributions go in the same direction as the previous ones: fixed-size populations limit and damage the behavior of GP while the use of dynamically sized populations is often beneficial. Furthermore, the models presented in the above quoted references have been successfully used by Silva (2008) to study and control bloat.

5.2 GP for Dynamic Optimization

Dynamic environments abound and offer particular challenges for all optimization and problem-solving methods. A well-known strategy for survival in dynamic environments is to adopt a population-based approach (Dempsey 2007). This allows a diversity of potential solutions to be maintained, which increases the likelihood that a sufficiently good solution exists at any point in time to ensure the survival of the population in the long term. Dynamic environments can exhibit changes in many different ways including the frequency and degree/size of change. The types of changes might range from relatively small smooth transitions to substantial perturbations in all aspects of the domain (Dempsey 2007).

Given the power of the biological process of evolution to adapt to ever, changing environments, it is surprising that the number of studies applying and explicitly studying their artificial counterpart of GP in dynamic environments have been minimal (Dempsey 2007). Despite the existence of a recent Special Issue in the GP Journal on Dynamic environments (Yang et al. 2006), none of the four articles actually dealt with GP directly (e.g., Wang and Wineberg (2006)). While some applications in dynamic environments have been undertaken in the past two years (e.g., Wagner et al. 2007, Hansen et al. 2007, Jakobović and Budin 2006, and Kibria and Li 2006), there has been little analysis of the behavior of GP in these environments. The two main examples have examined bloat (Langdon and Poli 1998) and constant generation (Dempsey 2007).

This seems to hint that one is missing the boat by focusing on static problems where there is a single fixed target. Recent experiments in evolutionary biology simulations suggest that EC/evolution could work efficiently "because" of dynamic environments as opposed to "despite of" them (Kashtan et al. 2007). Au1

5.3 Generalization in GP

Generalization is one of the most important performance-evaluation criteria for artificial learning systems. Many results exist concerning the issue of generalization in machine learning, like, for instance, support vector machines (see for instance Smola and Scholkopf 1999). However, this issue in GP has not received the attention it deserves and only recently a few papers dealing with the problem of generalization have appeared (Eiben and Jelasity 2002; Kushchu 2002). A detailed survey of the main contributions on generalization in GP has been done by Kushchu (2002). Another important contribution to the field of generalization in GP is due to the work of Banzhaf and coworkers. In particular, in Francone et al. (1996) they introduce a GP system called *Compiling GP* and they compared its generalization ability with that of other machine-learning paradigms. Furthermore, in Banzhaf et al. (1996) they show the positive effect of an extensive use of the mutation operator on generalization in GP using sparse data sets. In 2006, Da Costa and Landry (2006) have proposed a new GP model called *Relaxed GP*, showing its generalization ability. In 2006, Gagné et al. (2006) have investigated two methods to improve generalization in GP-based learning: (1) the selection of the best-of-run individuals using a three-data-sets methodology, and (2) the application of parsimony pressure to reduce the complexity of the solutions.

A common design principle among ML researchers is the so called *minimum description length principle* (see for instance Rissanen 1978), which states that the best model is the one that minimizes the amount of information needed to encode it. In this perspective, preference for simpler solutions and over-fitting avoidance seem to be closely related. It is more likely that a complex solution incorporates specific information from the training set, thus over-fitting it, compared to a simpler solution. But, as mentioned in Domingos (1999), this argumentation should be taken with care as too much emphasis on minimizing complexity can prevent the discovery of more complex yet more accurate solutions. Finally, in Vanneschi et al. (2007), authors hint that GP generalization ability (or more precisely, *extrapolation* ability) can be improved by performing a multi-optimization on the training set. A related idea has been used in Gagné et al. (2006) in the domain of binary classification, where a two-objective sort is performed in order to extract a set of nondominated individuals. However, different objectives in the realm of regression problems are used in Vanneschi et al. (2007).

Despite the above mentioned studies, the issue of generalization remains an open one in GP, since no theoretical approach has ever been presented to tackle this problem until now.

5.4 Modularity in GP

The use of modules may be very important to improve GP expressiveness, code reusability, and performance. The best-known of these methods is Koza's ADFs, which is introduced in Sect. 2.7. The first step toward a theoretically motivated study of ADFs is probably represented by Rosca (1995), where an algorithm for the automatic discovery of building blocks in GP called *Adaptive Representation Through Learning* was presented. In the same year, Spector (1995) introduced techniques to evolve collections of automatically defined macros and showed how they can be used to produce new control structures during the evolution of programs and Seront (1995) extended the concept of code reuse in GP to the concept of generalization, showing how programs (or "concepts," using Seront's terminology) synthesized by GP to solve a problem can be reused to solve other ones. Altenberg offers a critical analysis of modularity in evolution in Altenberg (2004), stating that the evolutionary advantages that have been attributed to modularity do not derive from modularity *per se*. Rather, they require that there be an "alignment" between the spaces of phenotypic variation, and the selection gradients that are available to the organism. Modularity in the genotype–phenotype map may make such an alignment more readily attained, but it is not sufficient; the appropriate phenotype-fitness map in conjunction with the genotype–phenotype map is also necessary for evolvability. This contribution is interesting and stimulating, but its applicability to GP remains an open question.

In Jonyer and Himes (2006) the concept of ADFs is extended by using graph-based data mining to identify common aspects of highly fit individuals and modularizing them by creating functions out of the subprograms identified. In Hemberg et al. (2007) the authors state that the ability of GP to scale to problems of increasing difficulty operates on the premise that it is possible to capture regularities that exist in a problem environment by decomposition of the problem into a hierarchy of modules.

Although the use of modularity in GP has helped solve some problems and provide new data/control abstractions, for instance in the form of ADFs, still some issues remain open. For instance: Has it solved really substantially more complex problems and does it hold the promise to do so in the near future? Are ADFs necessary/sufficient as a formalism to help solve grand-challenge problems, that is to provide scalability? And even more ambitiously: Can one achieve modularity, hierarchy, and reuse in a more controlled and principled manner? Can one provide insights and metrics on how to achieve this? Is it possible to routinely evolve algorithms? How to achieve iteration and/or recursion in a controlled and principled manner? What is the best way to handle the generation and maintenance of constants?

A first attempt to answer this last question is represented by Vanneschi et al. (2006) and Cagnoni et al. (2005) where the authors present a new coevolutionary environment where a GA dynamically evolves constants for GP. Although interesting, this contribution, as many others aimed at improving GP modularity, is mainly empirical, that is, it shows the benefits of the presented method(s) by means of a set of experimental results on some test functions. In order to be able to get a deeper insight of the real usefulness of modularity and code-reuse in GP, more theoretical studies are needed.

6 Conclusions

GP is a systematic method for getting computers to automatically solve a problem starting from its high-level specifications. It extends the model of genetic algorithms to the space of computer programs. In its almost 30 years of existence, it has been successfully applied to a number of challenging real-world problem domains. Furthermore, solid theory exists explaining its behavior and dynamics.

After a general introduction to the mechanics of GP, this chapter has discussed some examples of real-world applications of GP. In general, GP is particularly suitable for applications of curve fitting, data modeling, or symbolic regression, typically characterized by a large amount of data and where little is known about the relationships among the relevant features. Over the years, dozens of GP results have passed the human-competitiveness test, defined by Koza by establishing eight precise criteria, including patentability, beating humans in regulated competitions, producing publishable results, etc. Fields where GP have been successfully applied in the last few years include: image and signal processing; financial trading, time series, and economic modeling; industrial process control; medicine, biology and bioinformatics; computational chemistry; entertainment and computer games; arts; and compression.

Subsequently, some of the most relevant GP theoretical results, including schema theories and Markov chains-based models have been presented. Recent studies of the distribution of fitness in GP search spaces and of the distribution of programs' length have been discussed. Various theories explaining the phenomenon of bloat, including the recently introduced crossover bias theory, which is based on a precise equation of the programs' size evolution, and that, hopefully, will give a conclusive explanation of this phenomenon have been presented. Arguments in support of the conjecture that GP is a global optimiser, that is, it is guaranteed to find a globally optimal solution, given enough time, have been given. Furthermore, the most important results obtained in the field of GP problem difficulty, including a discussion of fitness distance correlation and negative slope coefficient, two of the most effective measures of problem difficulty for GP, and the recently introduced subtree crossover similarity/dissimilarity measure, that should help in defining new and hopefully more powerful difficulty measures have been presented. Finally, results on the existence of a "free lunch" for GP under some weak (and, in practice, often satisfied) conditions have been summarized.

In the last part of this chapter, hot topics and open issues of GP have been discussed, including the development of a self-adapting framework to dynamically modify the GP configuration (population size, representation, genetic operators, etc.) during the evolution; the study of the usefulness of GP for dynamic optimization problems; the study and better understanding of the concept of generalization and its implications for GP; and the definition of more powerful and self-adaptive methods of using modularity in GP.

Although this chapter could not exhaustively cover the immensity of the field, it is hoped that this chapter will attract newcomers to the field of GP as well as help focus future research, leading to a wider application and deeper understanding of GP as a problem-solving strategy.

Reference

Altenberg L (2004) Modularity in evolution: Some low-level questions. In: Rasskin-Gutman D, Callebaut W (eds), *Modularity: Understanding the Development and Evolution of Complex Natural Systems*. MIT Press, Cambridge, MA, In press.

Alves da Silva A P, Abrao P J (2002) Applications of evolutionary computation in electric power systems. In: Fogel D B et al. (eds), *Proceedings of the 2002 Congress on Evolutionary Computation CEC2002*, pp. 1057–1062. IEEE Press

Archetti F, Messina E, Lanzeni S, Vanneschi L (2007) Genetic programming for computational pharmacokinetics in drug discovery and development. *Genet Programming Evol Mach* 8(4):17–26

Azaria Y, Sipper M (2005) GP-gammon: Genetically programming backgammon players. *Genet Programming Evol Mach* 6(3):283–300, Sept. Published online: 12 August 2005.

Banzhaf W, Francone F D, Nordin P (1996) The effect of extensive use of the mutation operator on generalization in genetic programming using sparse data sets. In: Ebeling W et al., (ed) *4th International Conference on Parallel Problem Solving from Nature (PPSN96)*, Springer, Berlin, pp. 300–309.

Banzhaf W, Nordin P, Keller RE, Francone FD (1998) *Genetic programming, An Introduction*. Morgan Kaufmann, San Francisco, CA

Barrett S J, Langdon W B (2006) Advances in the application of machine learning techniques in drug discovery, design and development. In: Tiwari A et al. (eds), *Applications of Soft Computing: Recent Trends*, Advances in Soft Computing, On the World Wide Web, 19 Sept. – 7 Oct. 2005. Springer, Berlin, 99–110.

Bojarczuk C C, Lopes H S, Freitas A A (July–Aug. 2008) Genetic programming for knowledge discovery in chest-pain diagnosis. *IEEE Eng Med Biol Mag* 19 (4):38–44.

Brameier M, Banzhaf W (2001) A comparison of linear genetic programming and neural networks in medical data mining. *IEEE Trans Evol Comput* 5(1):17–26

Cagnoni S, Rivero D, Vanneschi L (2005) A purely-evolutionary memetic algorithm as a first step towards symbiotic coevolution. In: *Proceedings of the 2005 IEEE Congress on Evolutionary Computation (CEC'05)*, Edinburgh, Scotland, 2005. IEEE Press, Piscataway, NJ. pp. 1156 – 1163

Castillo F, Kordon A, Smits G (2006) Robust pareto front genetic programming parameter selection based on design of experiments and industrial data. In: Riolo R L, et al. (ed) *Genetic Programming Theory and Practice IV*, vol 5 of *Genetic and Evolutionary Computation*, chapter 2, pages –. Springer, Ann Arbor, 11–13 May.

Chen S-H, Liao C-C (2005) Agent-based computational modeling of the stock price-volume relation. *Inf Sci* 170(1):75–100, 18 Feb.

Chen S-H, Wang H-S, Zhang B-T (1999) Forecasting high-frequency financial time series with evolutionary neural trees: The case of heng-sheng stock index. In: Arabnia H R, (ed), *Proceedings of the International Conference on Artificial Intelligence, IC-AI '99*, vol 2, Las Vegas, NV, 28 June-1 July. CSREA Press pp. 437–443.

Da Costa LE, Landry JA (2006) Relaxed genetic programming. In: Keijzer M et al., editor, *GECCO 2006: Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, vol 1, Seattle, Washington, DC, 8–12 July. ACM Press pp. 937–938

Dassau E, Grosman B, Lewin D R (2006) Modeling and temperature control of rapid thermal processing. *Comput Chem Eng* 30(4):686–697, 15 Feb.

Davis T E, Principe J C (1993) A Markov chain framework for the simple genetic algorithm. *Evol Comput* 1(3):269–288.

De Jong KA (1988) Learning with genetic algorithms: An overview. Mach Learn 3:121–138

Dempsey I (2007) Grammatical evolution in dynamic environments. Ph.D. thesis, University College Dublin, Ireland.

Dignum S, Poli R (2007) Generalisation of the limiting distribution of program sizes in tree-based genetic programming and analysis of its effects on bloat. In: Thierens, D et al. (eds), *GECCO '07: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, vol 2 London, 7–11 July 2007. ACM Press, pp. 1588–1595

Eiben AE, Jelasity M (2002) A critical note on experimental research methodology in EC. In: *Congress on*

Evolutionary Computation (CEC'02), Honolulu, HI, 2002. IEEE Press, Piscataway, NJ, pp. 582–587

Esparcia-Alcazar A I, Sharman K C (Sept. 1996) Genetic programming techniques that evolve recurrent neural networks architectures for signal processing. In: *IEEE Workshop on Neural Networks for Signal Processing*, Seiko, Kyoto, Japan

Fernandez F, Martin A (2004) Saving effort in parallel GP by means of plagues. In: Keijzer M, et al. (eds), *Genetic Programming 7th European Conference, EuroGP 2004, Proceedings*, vol 3003 of *LNCS*, Coimbra, Portugal, 5–7 Apr. Springer-Verlag, pp. 269–278

Fernandez F, Tomassini M, Vanneschi L (2003) Saving computational effort in genetic programming by means of plagues. In: Sarker, R et al. (eds), *Proceedings of the 2003 Congress on Evolutionary Computation CEC2003*, Camberra, 8–12 Dec. 2003. IEEE Press, pp. 2042–2049

Fernandez F, Vanneschi L, Tomassini M (2003) The effect of plagues in genetic programming: A study of variable-size populations. In: Ryan, C et al. (ed) *Genetic Programming, Proceedings of EuroGP'2003*, vol 2610 of *LNCS*, Essex, 14–16 Apr. Springer-Verlag, pp. 317–326

Forrest S, Mitchell M (1993) What makes a problem hard for a genetic algorithm? Some anomalous results and their explanation. *Mach Learn* 13:285–319

Francone FD, Nordin P, Banzhaf W (1996) Benchmarking the generalization capabilities of a compiling genetic programming system using sparse data sets. In: Koza J R et al. (ed), *Genetic Programming: Proceedings of the First Annual Conference*, MIT Press, Cambridge, pp. 72–80

Fukunaga A, Stechert A (1998) Evolving nonlinear predictive models for lossless image compression with genetic programming. In: Koza, J R et al. (eds), *Genetic Programming 1998: Proceedings of the Third Annual Conference*, University of Wisconsin, Madison, WI, 22–25 July Morgan Kaufmann pp. 95–102

Gagné C, Schoenauer M, Parizeau M, Tomassini M (2006) Genetic programming, validation sets, and parsimony pressure. In: Collet P et al. (ed), *Genetic Programming, 9th European Conference, EuroGP2006*, Lecture Notes in Computer Science, LNCS 3905, pp. 109–120. Springer, Berlin, Heidelberg, New York

Giacobini M, Tomassini M, Vanneschi L (2002) Limiting the number of fitness cases in genetic programming using statistics. In: Merelo JJ, et al. (eds), *Parallel Problem Solving from Nature – PPSN VII*, vol 2439 of Lecture Notes in Computer Science, Springer-Verlag, Heidelberg, pp. 371–380

Goldberg DE (1989) *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, Boston, MA.

Gustafson S, Vanneschi L (2005) Operator-based distance for genetic programming: Subtree crossover distance. In: Keijzer, M., et al. (ed), *Genetic Programming, 8th European Conference, EuroGP2005*, Lecture Notes in Computer Science, LNCS 3447, pp. 178–189. Springer, Berlin, Heidelberg, New York

Gustafson S, Vanneschi L (2008) Operator-based tree distance in genetic programming. *IEEE Trans Evol Comput* 12:4

Hansen J V, Lowry P B, Meservy R D, McDonald D M (Aug. 2007) Genetic programming for prevention of cyberterrorism through dynamic and evolving intrusion detection. *Decis Support Syst* 43(4):1362–1374, Special Issue Clusters.

Hasan S, Daugelat S, Rao P S S, Schreiber M (June 2006) Prioritizing genomic drug targets in pathogens: Application to mycobacterium tuberculosis. *PLoS Comput Biol* 2(6):e61

Hauptman A, Sipper M (2007) Evolution of an efficient search algorithm for the mate-in-N problem in chess. In: Ebner, M et al. (eds), *Proceedings of the 10th European Conference on Genetic Programming*, vol 4445 of Lecture Notes in Computer Science Valencia, Spain, 11 – 13 Apr. Springer pp. 78–89

Hemberg E, Gilligan C, O'Neill M, Brabazon A (2007) A grammatical genetic programming approach to modularity in genetic algorithms. In: Ebner, M et al. (eds), *Proceedings of the 10th European Conference on Genetic Programming*, vol 4445 of Lecture Notes in Computer Science, Valencia, Spain, 11 – 13 Apr. Springer pp. 1–11.

Holland JH (1975) *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, Ann Arbor, MI

Horn J, Goldberg DE (1995) Genetic algorithm difficulty and the modality of the fitness landscapes. In: Whitley D, Vose M (eds), *Foundations of Genetic Algorithms*, vol. *3*, Morgan Kaufmann, pp. 243–269

Howard D, Roberts S C (2004) Incident detection on highways. In: O'Reilly, U-M et al., (eds), *Genetic Programming Theory and Practice II*, chapter 16, Springer, Ann Arbor, 13–15 May pp. 263–282

Jacob C (May–June 2000) The art of genetic programming. *IEEE Intell Syst* 15(3):83–84, May–June

Jacob C (2001) *Illustrating Evolutionary Computation with Mathematica*. Morgan Kaufmann, San Francisco, CA

Jaković D, Budin L (2006) Dynamic scheduling with genetic programming. In: Collet, P et al. (eds), *Proceedings of the 9th European Conference on Genetic Programming*, vol 3905 of Lecture Notes in Computer Science, Budapest, Hungary, 10 – 12 Apr. Springer pp. 73–84

Jin N, Tsang E (2006) Co-adaptive strategies for sequential bargaining problems with discount factors and outside options. In: *Proceedings of the 2006 IEEE*

Congress on Evolutionary Computation, Vancouver, 6–21 July. IEEE Press pp. 7913–7920

Jones T (1995) Evolutionary algorithms, fitness landscapes and search. Ph.D. thesis, University of New Mexico, Albuquerque

Jonyer I, Himes A (2006) Improving modularity in genetic programming using graph-based data mining. In: Sutcliffe G C J Goebe R G (eds), *Proceedings of the Nineteenth International Florida Artificial Intelligence Research Society Conference*, pp. 556–561, Melbourne Beach, FL, May 11–13 2006. American Association for Artificial Intelligence.

Jordaan E, den Doelder J, Smits G (2006) Novel approach to develop structure-property relationships using genetic programming. In: Runarsson T P, et al. (eds), *Parallel Problem Solving from Nature – PPSN IX*, vol 4193 of *LNCS*, Reykjavik, Iceland, 9–13 Sept. Springer-Verlag pp. 322–331

Kashtan N, Noor E, Alon U (2007) Varying environments can speed up evolution. *Proceedings of the National Academy of Sciences*, 104(34):13711–13716, August 21

Kattan A, Poli R (2008) Evolutionary lossless compression with GP-ZIP. In *Proceedings of the IEEE World Congress on Computational Intelligence*, Hong Kong, 1–6 June. IEEE. forthcoming.

Keijzer M (Sept. 2004) Scaled symbolic regression. *Genetic Programming and Evolvable Machines*, 5(3):259–269

Kibria R H, Li Y (2006) Optimizing the initialization of dynamic decision heuristics in DPLL SAT solvers using genetic programming. In: Collet P, et al. (eds), *Proceedings of the 9th European Conference on Genetic Programming*, vol. 3905 of Lecture Notes in Computer Science, Budapest, Hungary, 10 – 12 Apr. Springer. pp. 331–340

Kinnear KE Jr (1994) Fitness landscapes and difficulty in genetic programming. In: *Proceedings of the First IEEEConference on Evolutionary Computing*, IEEE Press, Piscataway, NY, pp. 142–147

Kirchherr W, Li M, Vitanyi P (1997) The miraculous universal distribution. *Math Intell* 19:7–15

Klappenecker A, May F U (1995) Evolving better wavelet compression schemes. In: Laine, A F et al. (ed), *Wavelet Applications in Signal and Image Processing III*, vol 2569, San Diego, CA 9–14 July. SPIE.

Kordon A (Sept. 2006) Evolutionary computation at dow chemical. *SIGEVOlution*, 1(3):4–9

Koza J, Poli R (2003) A genetic programming tutorial. In: Burke E (ed) *Introductory Tutorials in Optimization, Search and Decision Support*, Chapter 8. http://www.genetic-programming.com/jkpdf/burke2003 tutorial.pdf

Koza J R (1992a) A genetic approach to the truck backer upper problem and the inter-twined spiral problem. In *Proceedings of IJCNN International Joint Conference on Neural Networks*, vol IV, IEEE Press, pp. 310–318.

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

Koza JR (1994) *Genetic Programming II*. The MIT Press, Cambridge, MA

Koza J R, Bennett F H III, Stiffelman O (1999) Genetic programming as a Darwinian invention machine. In: Poli R, et al. (eds) *Genetic Programming, Proceedings of EuroGP'99*, vol 1598 of *LNCS*, Goteborg, Sweden, 26–27 May. Springer-Verlag pp. 93–108.

Koza JR, Bennett FH III, Andre D, Keane MA (1999) *Genetic Programming III: Darwinian Invention and Problem Solving*. Morgan Kaufmann, San Francisco, CA

Kushchu I (2002) An evaluation of evolutionary generalization in genetic programming. Artif Intell Rev 18(1):3–14

Langdon WB (2003) Convergence of program fitness landscapes. In: Cantú-Paz, E., *et al.* (ed) *Genetic and Evolutionary Computation – GECCO-2003*, vol 2724 of *LNCS*, Springer-Verlag, Berlin, pp. 1702–1714

Langdon W B, Buxton B F (Sept. 2004) Genetic programming for mining DNA chip data from cancer patients. *Genet Programming Evol Mach*, 5 (3):251–257

Langdon W B, Poli R (1998) Genetic programming bloat with dynamic fitness. In: Banzhaf W, et al. (eds), *Proceedings of the First European Workshop on Genetic Programming*, vol 1391 of *LNCS*, Paris, 14–15 Apr. Springer-Verlag. pp. 96–112

Langdon W B, Poli R (2002) *Foundations of Genetic Programming*. Springer-Verlag

Langdon W B, Poli R (2005) Evolutionary solo pong players. In: Corne, D et al. (eds), *Proceedings of the 2005 IEEE Congress on Evolutionary Computation*, vol 3, Edinburgh, U.K., 2–5 Sept. IEEE Press pp. 2621–2628

Langdon W B, Soule T, Poli R, Foster J A (June 1999) The evolution of size and shape. In Spector, L et al. (eds), *Advances in Genetic Programming 3*, chapter 8, pp. 163–190. MIT Press, Cambridge, MA.

Lew T L, Spencer A B, Scarpa F, Worden K, Rutherford A, Hemez F (Nov. 2006) Identification of response surface models using genetic programming. *Mech Syst Signal Process* 20(8):1819–1831

Lewin D R, Lachman-Shalem S, Grosman B (July 2006) The role of process system engineering (PSE) in integrated circuit (IC) manufacturing. *Control Eng Pract* 15(7):793–802 Special Issue on Award Winning Applications, 2005 IFAC World Congress.

Louchet J (June 2001) Using an individual evolution strategy for stereovision. *Genet Programming Evol Mach* 2(2):101–109

Lutton E, Levy-Vehel J, Cretin G, Glevarec P, Roll C (1995) Mixed IFS: Resolution of the inverse problem using genetic programming. Research Report No 2631, Inria

Machado P, Romero J (eds). (2008) *The Art of Artificial Evolution*. Springer, Forthcoming

McPhee N F, Miller J D (1995) Accurate replication in genetic programming. In: Eshelman L (ed), *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, Pittsburgh, PA 15–19 July Morgan Kaufmann pp. 303–309

Miller J (2001) What bloat? Cartesian genetic programming on boolean problems. In: Goodman ED (ed), *2001 Genetic and Evolutionary Computation Conference Late Breaking Papers*, pp. 295–302, San Francisco, CA 9–11 July

Mitavskiy B, Rowe J (2006) Some results about the markov chains associated to GPs and to general EAs. *Theor Comput Sci* 361(1):72–110 28 Aug

Mitchell M, Forrest S, Holland J (1992) The royal road for genetic algorithms: Fitness landscapes and ga performance. In: Varela FJ, Bourgine P (eds), *Toward a Practice of Autonomous Systems, Proceedings of the First European Conference on Artificial Life*, The MIT Press, pp. 245–254

Nikolaev NI, Slavov V (1998) Concepts of inductive genetic programming. In: Banzhaf, W., et al. (ed), *Genetic Programming, Proceedings of EuroGP'1998*, vol 1391 of *LNCS*, Springer-Verlag, pp. 49–59

Nordin P, Banzhaf W (1996) Programmatic compression of images and sound. In: Koza J R, et al. (eds), *Genetic Programming 1996: Proceedings of the First Annual Conference*, Stanford University, CA 28–31 July. MIT Press pp. 345–350

Domingos P (1999) The role of Occam's razor in knowledge discovery. *Data Mining Knowl Discov* 3(4):409–425

Poli R (1996) Genetic programming for image analysis. In: Koza J R et al. (eds), *Genetic Programming 1996: Proceedings of the First Annual Conference*, Stanford University, CA 28–31 July MIT Press pp. 363–368

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

Poli R, Langdon WB (1997) Genetic programming with one-point crossover and point mutation. Tech. Rep. CSRP-97-13, University of Birmingham, B15 2TT, U.K., 15

Poli R, Langdon W B (2006) Efficient markov chain model of machine code program execution and halting. In: Riolo R L, et al. (eds), *Genetic Programming Theory and Practice IV*, vol 5 of *Genetic and Evolutionary Computation*, chapter 13. Springer, Ann Arbor, 11–13 May

Poli R, Langdon W B, Dignum S (2007) On the limiting distribution of program sizes in tree-based genetic programming. In: Ebner, M et al. (eds), *Proceedings of the 10th European Conference on Genetic Programming*, vol 4445 of Lecture Notes in Computer Science, Valencia, Spain, 11 – 13 Apr. Springer pp. 193–204

Poli R, McPhee N F (Mar. 2003a) General schema theory for genetic programming with subtree-swapping crossover: Part I. *Evol Comput* 11(1):53–66

Poli R, McPhee N F (June 2003b) General schema theory for genetic programming with subtree-swapping crossover: Part II. *Evol Comput* 11(2):169–206

Poli R, McPhee NF (2008) Parsimony pressure made easy. In: *GECCO '08: Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation*, pp. 1267–1274, New York, NY, ACM.

Poli R, McPhee N F, Rowe J E (Mar. 2004) Exact schema theory and markov chain models for genetic programming and variable-length genetic algorithms with homologous crossover. *Genet Programming Evol Mach* 5(1):31–70.

Poli R, McPhee NF, Graff M (2009) Free lunches for symbolic regression. In: *Foundations of Genetic Algorithms (FOGA)*. ACM, forthcoming.

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

Rissanen J (1978) Modeling by shortest data description. *Automatica* 14:465–471

Rochat D, Tomassini M, Vanneschi L (2005) Dynamic size populations in distributed genetic programming. In: Keijzer M, et al. (eds), *Proceedings of the 8th European Conference on Genetic Programming*, vol 3447 of Lecture Notes in Computer Science, Lausanne, Switzerland, 30 Mar. – 1 Apr. Springer. pp. 50–61

Rosca JP (1995) Towards automatic discovery of building blocks in genetic programming. In: *Working Notes for the AAAI Symposium on Genetic Programming*, AAAI, pp. 78–85.

Rudolph G (1994) Convergence analysis of canonical genetic algorithm. *IEEE Trans Neural Netw* 5(1):96–101

Rudolph G (1996) Convergence of evolutionary algorithms in general search spaces. In: *International Conference on Evolutionary Computation*, pp. 50–54

Schumacher C, Vose MD, Whitley LD (2001) The no free lunch and problem description length. In: *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, Morgan Kaufmann, pp. 565–570

Seront G (1995) External concepts reuse in genetic programming. In: Siegel E V, Koza J R (eds), *Working Notes for the AAAI Symposium on Genetic Programming*, MIT, Cambridge, MA 10–12 Nov. AAAI pp. 94–98,

Shah S C, Kusiak A (July 2004) Data mining and genetic algorithm based gene/SNP selection. *Artif Intell Med* 31(3):183–196

Silva S G O (2008) Controlling bloat: individual and population based approaches in genetic programming. Ph.D. thesis, Universidade de Coimbra, Faculdade de Ciences e Tecnologia, Departamento de Engenharia Informatica, Portugal

Smola A J, Scholkopf B (1999) A tutorial on support vector regression. Technical Report Technical Report Series – NC2-TR-1998-030, NeuroCOLT2

Soule T, Foster J A (1998a) Effects of code growth and parsimony pressure on populations in genetic programming. *Evol Comput* 6(4):293–309, Winter

Soule T, Foster J A (1998b) Removal bias: A new cause of code growth in tree based evolutionary programming. In *1998 IEEE International Conference on Evolutionary Computation*, Anchorage, Alaska 5–9 May IEEE Press. pp. 781–186

Spector L (1995) Evolving control structures with automatically defined macros. In: Siegel E V, Koza J R (eds), *Working Notes for the AAAI Symposium on Genetic Programming*, MIT, Cambridge, MA 10–12 Nov. AAAI pp. 99–105

Stadler PF (2002) Fitness landscapes. In: Lässig M, Valleriani (eds), *Biological Evolution and Statistical Physics*, vol 585 of Lecture Notes Physics, pp. 187–207, Heidelberg, Springer-Verlag.

Stephens C R, Waelbroeck H (1999) Schemata evolution and building blocks. *Evol Comput* 7(2):109–124

Tomassini M, Vanneschi L, Cuendet J, Fernandez F (2004) A new technique for dynamic size populations in genetic programming. In: *Proceedings of the 2004 IEEE Congress on Evolutionary Computation*, Portland, Oregon, 20–23 June. IEEE Press pp. 486–493

Tomassini M, Vanneschi L, Collard P, Clergue M (2005) A study of fitness distance correlation as a difficulty measure in genetic programming. *Evol Comput* 13(2):213–239, Summer

Trujillo L, Olague G (2006) Using evolution to learn how to perform interest point detection. In: X Y T et al. (ed), *ICPR 2006 18th International Conference on Pattern Recognition*, vol 1, IEEE, pp. 211–214. 20–24 Aug.

Tsang E Jin N (2006) Incentive method to handle constraints in evolutionary. In: Collet P, et al. (eds), *Proceedings of the 9th European Conference on Genetic Programming*, vol 3905 of Lecture Notes in Computer Science, Budapest, Hungary, 10–12 Apr. Springer. pp. 133–144

Vanneschi L (2004) Theory and practice for efficient genetic programming Ph.D. thesis, Faculty of Sciences, University of Lausanne, Switzerland

Vanneschi L (2007) Investigating problem hardness of real life applications. In: R. R. et al., (ed), *Genetic Programming Theory and Practice V*, Springer US, Computer Science Collection, pp. 107–124, Chapter 7.

Vanneschi L, Clergue M, Collard P, Tomassini M, Vérel S (2004) Fitness clouds and problem hardness in genetic programming. In: Deb K, et al. (eds), *Genetic and Evolutionary Computation – GECCO-2004, Part II*, vol 3103 of Lecture Notes in Computer Science Seattle, WA 26–30 June Springer-Verlag pp. 690–701,

Vanneschi L, Gustafson S, Mauri G (2006) Using subtree crossover distance to investigate genetic programming dynamics. In: Collet, P., et al. (ed), *Genetic Programming, 9th European Conference, EuroGP2006*, Lecture Notes in Computer Science, LNCS 3905, pp. 238–249. Springer, Berlin, Heidelberg, New York

Vanneschi L, Mauri G, Valsecchi A, Cagnoni S (2006) Heterogeneous cooperative coevolution: strategies of integration between GP and GA. In: Keijzer M, et al. (eds), *GECCO 2006: Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, vol 1, Seattle, Washington, DC, 8–12 July. ACM Press. pp. 361–368,

Vanneschi L, Rochat D, Tomassini M (2007) Multi-optimization improves genetic programming generalization ability. In: Thierens D, et al. (eds), *GECCO '07: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, vol 2, London, 7–11 July. ACM Press. pp. 1759–1759

Vanneschi L, Tomassini M, Collard P, Clergue M (2003) Fitness distance correlation in structural mutation genetic programming. In: Ryan, C., et al., (ed), *Genetic Programming, 6th European Conference, EuroGP2003*, Lecture Notes in Computer Science, Springer-Verlag, Heidelberg, pp 455–464

Vanneschi L, Tomassini M, Collard P, Vérel S (2006) Negative slope coefficient. A measure to characterize genetic programming. In: Collet P, et al. (eds), *Proceedings of the 9th European Conference on Genetic Programming*, vol 3905 of Lecture Notes in Computer Science, Budapest, Hungary, 10 – 12 Apr. Springer. pp. 178–189

Wagner N, Michalewicz Z, Khouja M, McGregor R R (Aug. 2007) Time series forecasting for dynamic environments: The dyfor genetic program model. *IEEE Trans Evol Comput* 11(4):433–452

Wang Y, Wineberg M (Dec. 2006) Estimation of evolvability genetic algorithm and dynamic environments. *Genet Programming Evol Mach* 7(4):355–382

Wedge DC, Kell DB (2008) Rapid prediction of optimum population size in genetic programming using a novel genotype – fitness correlation. In: Keijzer M, et al. (eds), *GECCO '08: Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation*, Atlanta, GA, ACM pp. 1315–1322

Whitley D, Watson JP (2005) Complexity theory and the no free lunch theorem. In: Burke EK, Kendall G (eds), *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, Chapter 11, pp. 317–339. Springer US

Wolpert D, Macready W (April 1997) No free lunch theorems for optimization. *IEEE Trans Evol Comput* 1(1):67–82

Woodward J R, Neil J R (2003) No free lunch, program induction and combinatorial problems. In: Ryan C, et al. (eds), *Genetic Programming, Proceedings of EuroGP'2003*, vol 2610 of *LNCS*, Essex, 14–16 Apr. Springer-Verlag pp. 475–484

Xie H, Zhang M, Andreae P (2006) Genetic programming for automatic stress detection in spoken english. In: Rothlauf F, et al. (eds), *Applications of Evolutionary Computing, EvoWorkshops 2006: EvoBIO, EvoCOMNET, EvoHOT, EvoIASP, EvoInteraction, EvoMUSART, EvoSTOC*, vol 3907 of *LNCS*, pp. 460–471, Budapest, 10–12 Apr. Springer Verlag.

Yang S, Ong Y-S, Jin Y (Dec. 2006) Editorial to special issue on evolutionary computation in dynamic and uncertain environments. *Genet Programming Evol Mach* 7(4):293–294, Editorial.

Yu T, Chen S-H (2004) Using genetic programming with lambda abstraction to find technical trading rules. In: *Computing in Economics and Finance*, University of Amsterdam, 8–10 July

Yu J, Yu J, Almal A A, Dhanasekaran S M, Ghosh D, Worzel W P, Chinnaiyan A M (Apr. 2007) Feature selection and molecular classification of cancer using genetic programming. *Neoplasia* 9(4):292–303

Zhang B-T, Mühlenbein H (1995) Balancing accuracy and parsimony in genetic programming. *Evol Comput* 3(1):17–38

Zhang M, Smart W (Aug. 2006) Using gaussian distribution to construct fitness functions in genetic programming for multiclass object classification. *Pattern Recog Lett* 27(11):1266–1274. Evolutionary Computer Vision and Image Understanding.