Chapter 8
Genetic Programming

GAs, studied in Chapter 3, are capable of solving many problems and simple enough to allow for solid theoretical studies. Nevertheless, the representation of individuals that characterizes GAs (i.e., the fact that individuals in GAs must be strings of a previously fixed length) can be a limitation for a wide set of applications. In these cases, the most natural representation for a solution is a hierarchical computer program, rather than a string of characters of a fixed length. For example, strings of a static length do not readily support the hierarchical organization of tasks into subtasks typical of computer programs, they do not provide any convenient way of incorporating iteration and recursion, and so on. But above all, GA representation schemes do not have any dynamic variability: the initial selection of string length limits in advance the number of internal states of the system and limits what the system can learn.

This lack of representation power (already recognized in [De Jong, 1988]) is overcome by Genetic Programming (GP) [Koza, 1992]. As with GAs, GP belongs to the field of Evolutionary Computation, but contrarily to GAs, GP operates with general hierarchical computer programs. In other words, GP manages a population of computer programs. These programs are typically initialized randomly, and then evolved, using an algorithm that is inspired by the Theory of Evolution of Charles Darwin, with the objective of automatically discovering programs that are appropriate for solving the problem at hand. So, GP is a method for automatically generating a program that solves a problem, starting from the high-level specifications of the problem itself and, in general, without any intervention from a human programmer. As studied in the previous chapters, data models are particular computer programs, and, as such, GP can be seen as a Machine Learning method. Even though every programming language (e.g., Pascal, Fortran, C, etc.) is capable of expressing and executing general computer programs, Koza chose the *LISP* (*LIS*t *P*rocessing) language to code GP individuals. The reasons for this choice can be summarized as follows:

- Both programs and data have the same form in LISP, so it is possible and convenient to treat a computer program as data in the genetic population.

© Springer Nature Switzerland AG 2023
L. Vanneschi and S. Silva, *Lectures on Intelligent Systems*, Natural Computing Series,
https://doi.org/10.1007/978-3-031-17922-8_8

- This common form of programs and data is a parse tree and this allows us in a simple way to decompose a structure into substructures (subtrees) to be manipulated by the genetic operators.
- LISP facilitates the programming of structures whose size and shape change dynamically and the handling of hierarchical structures.
- Many programming tools were commercially available for LISP.

In other words, GP, as originally defined by Koza, considers individuals as LISP-like tree structures. These structures are perfectly capable of capturing all the fundamental properties of modern programming languages. GP using this representation is called *tree-based GP*[1]. The tree-based representation of genomes, although the oldest and most commonly used, is not the only one that has been employed in GP. In particular, in recent decades, growing attention has been dedicated by researchers to linear genomes [Brameier and Banzhaf, 2001], graph genomes [Miller, 2001], stack-based [Spector and Robinson, 2002] and grammar-based GP [O'Neill and Ryan, 2003].

This chapter is structured as follows: Section 8.1 introduces the main definitions and characteristics of GP and of its operators, including specifications of its typical behaviors and an example of a simple run, described step by step. Section 8.2 presents the most significant theoretical studies that have been performed in GP, known as schema theory. Section 8.3 describes some common typical problems of GP, or GP benchmarks, while Section 8.4 briefly describes some of the current challenges and open issues in GP. Finally, Sections 8.5 and 8.6 present two of the most recent and promising developments of GP.

8.1 The GP Process

In synthesis, the GP paradigm breeds computer programs to solve problems by executing the following steps:

1. Generate an initial population of computer programs (or individuals).
2. Iteratively perform the following steps until the termination criterion has been satisfied:

   2.1. Execute each program in the population and assign it a fitness value according to how well it solves the problem.
   2.2. Create a new population by applying the following operations:
      2.2.1. Probabilistically select a set of computer programs to be reproduced, on the basis of their fitness (selection).
      2.2.2. Copy some of the selected individuals, without modifying them, into the new population (replication[2]).

[1] Of course, modern tree-based GP is implemented in languages like C, C++, Java or Python, rather than LISP.

[2] Originally called reproduction in [Koza, 1992]

2.2.3. Create new computer programs by genetically recombining randomly chosen parts of two selected individuals (crossover).
2.2.4. Create new computer programs by substituting randomly chosen parts of some selected individuals with new randomly generated ones (mutation).
2.2.5. Between the old and new programs, select the ones that will form the new population (survival).

3. The best computer program that appears in any generation is designated as the result of the GP process at that generation. This result may be a solution (or an approximate solution) to the problem.

The attentive reader will not have failed to recognize clear similarities between the general functioning of GAs (presented in Chapter 3) and that of GP. The high-level algorithm is actually extremely similar, with the only major difference typically being that, in GP, the individuals resulting from the application of one genetic operator are not used as input for another genetic operator. In other words, crossover and mutation are not applied sequentially. More specifically, mutation is not applied to the offspring generated by crossover. Instead, some selected individuals undergo crossover, some of them mutation and some of them replication, according to fixed probabilities that are parameters of the algorithm. This is justified by the fact that, as we will study later in this chapter, GP genetic operators are more "disruptive" than those of GAs, and thus applying more than one operator would risk generating individuals that are too different from their parents. Besides this, the different representation of the individuals evolving in the population makes GP deeply different from GAs in its dynamics and interpretation. In the following sections, each part of the GP process is analyzed in detail, specifying the differences and similarities between GP and GAs.

8.1.1 Representation of GP Individuals

In tree-based GP, the set of all the possible structures that can be generated is the set of all the possible trees that can be built recursively from a set of function symbols $\mathscr{F} = \{f_1, f_2, \ldots, f_n\}$ (also called primitive functions, and used to label internal tree nodes) and a set of terminal symbols $\mathscr{T} = \{t_1, t_2, \ldots, t_m\}$ (used to label tree leaves). This potentially infinite search space is usually limited in size, by restricting it to only those trees whose depth is less than or equal to a previously fixed parameter d[3]. Each element in the function set $\mathscr{F}$ takes a fixed number of arguments, specifying its *arity*. Functions may include arithmetic operations ($+$, $-$, $*$, etc.), other 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/or any other domain-specific functions that may

[3] We recall that the *depth* of a tree is defined as the longest possible path from the root of the tree to one of its leaves.

be defined. Each terminal is typically either a variable or a constant, defined on the problem domain.

Example 8.1. Given the following sets of functions and terminals:

$$\mathscr{F} = \{+,-\}, \;\; \mathscr{T} = \{x,1\}$$

a legal GP individual is represented in Figure 8.1.

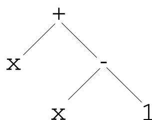

Fig. 8.1 A tree that can be built with the sets $\mathscr{F} = \{+,-\}$ and $\mathscr{T} = \{x,1\}$

This tree can also be represented by the following LISP-like S-expression (for a definition of LISP S-expressions see, for instance, [Koza, 1992]):

$$(+\, x\, (-\, x \;\; 1))$$

which is the prefix notation representation for expression $x+(x-1)$.

Closure and Sufficiency of Function Set and Terminal Set. The function and terminal sets should be chosen so as to verify the requirements of *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 be assumed by any terminal in the terminal set. In other words, each function should be well defined for any combination of arguments that it may encounter. The reason why this property must be verified is clearly that programs must be executed in order to assign them a fitness, and a failure in one of the executions of one of the programs composing the population would lead either to a failure of the whole GP system or to the generation of unpredictable results. The function and terminal sets of the previous example clearly satisfy the closure property. The following ones, for instance, don't satisfy this property:

$$\mathscr{F} = \{*,/\}, \;\; \mathscr{T} = \{x,0\}$$

In fact, each evaluation of an expression containing an operation of division by zero would cause an error. In order to use division, but avoid this type of problem, this operation is usually modified in order to verify the closure property (see Section 8.3.1). Respecting the closure property in real-life applications is not always straightforward, since the use of many different data types could be necessary. A common example is a mix of Boolean and numeric functions: function sets could exist composed of Boolean functions (*AND*, *OR*, ...), arithmetic functions ($+$, $-$,

$*, /, \ldots$), comparison functions ($>, <, =, \ldots$), conditionals (*IF THEN ELSE*), etc. and one might want to evolve expressions such as:

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

In such cases, introducing typed functions into the GP genome can help to force the closure property to be verified. GP in which each node carries its type as well as the types it can call, thus forcing functions calling it to cast the argument into the appropriate type, is called *strongly typed GP* [Banzhaf et al., 1998]. Using types might make even more sense in GP than with a human programmer, since a human programmer has a mental model of what needs to be done, whereas the GP system is completely random in its initialization and variation phases. Furthermore, type checking reduces the search space, which is likely to facilitate 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, the function and terminal sets of the previous example verify the sufficiency property if the problem at hand is an arithmetic one. It does not verify this property, for instance, if the problem faced is a logic one. For many domains, the requirements for sufficiency in the function and terminal sets are not clear and the definition of appropriate sets depends on the experience and the knowledge of the problem of the GP designer.

8.1.2 Initialization of a GP Population

Initialization of the population is the first step of the evolution process. It consists in 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, 1992]. These methods will be explained in the following paragraphs, where the set of function symbols composing the trees will be denoted by $\mathscr{F}$, the set of terminal symbols by $\mathscr{T}$ and the maximum depth allowed for the trees by d.

Grow initialization. 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 $\mathscr{F}$ to be the root of the tree;
- let n be the arity of the selected function symbol. Then n nodes are selected with uniform probability from the set $\mathscr{F} \cup \mathscr{T}$ to be its sons;
- for each function symbol between these n nodes, the method is recursively applied, i.e., its sons are selected from the set $\mathscr{F} \cup \mathscr{T}$, unless this symbol has a depth equal to $d - 1$. In the latter case, its sons are selected from $\mathscr{T}$.

In other words, the root is selected with uniform probability from $\mathscr{F}$, so that no tree composed of a single node is created initially (even though, in some implementations, trees composed of one single node can be admitted in the initial population).

Nodes with depths between 1 and $d-1$ are selected with uniform probability from $\mathscr{F} \cup \mathscr{T}$, but once a branch contains a terminal node, that branch has ended, even if the maximum depth d has not been reached. Finally, nodes at depth d are chosen with uniform probability from $\mathscr{T}$. Since the incidence of choosing terminals from $\mathscr{F} \cup \mathscr{T}$ is random throughout initialization, trees initialized using the grow method are likely to have irregular shape, i.e., to contain branches of various different lengths.

Full initialization. Full initialization works like grow initialization, with the difference that, instead of selecting nodes from $\mathscr{F} \cup \mathscr{T}$, the full method chooses only function symbols (i.e., symbols in $\mathscr{F}$) until a node is at a depth equal to $d-1$. Then it chooses only terminals (i.e., symbols in $\mathscr{T}$). The result is that every branch of the tree goes to the full maximum depth.

Ramped half-and-half initialization. As first noted by Koza [Koza, 1992], a population initialized with the above two methods may be composed of trees that are too similar to each other. In order to increase population diversity, the ramped half-and-half technique has been developed. Let d be the maximum-depth parameter. The population is divided equally among individuals to be initialized with trees having maximum depths equal to 1, 2, ..., $d-1$, d. For each depth group, half of the trees are initialized with the full technique and half with the grow technique.

8.1.3 Fitness Evaluation

Each program in the population is assigned a fitness value, representing its ability to solve the problem. This value is calculated by means of some well-defined explicit procedure. The two fitness measures most commonly used in GP are raw fitness and standardized fitness. They are described below.

Raw Fitness. Raw fitness, as defined by Koza, is "the measurement of fitness that is stated in the natural terminology of the problem itself". In other words, raw fitness is the most simple and natural way to calculate the ability of a program to solve a problem. For example, if the problem consists in driving a robot to make it pick up the maximum possible number of objects contained in a room, the raw fitness of a program that drives the robot could be the number of objects effectively picked up after 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 a polynomial, like for instance $x^4+x^3+x^2+x$, over the set of natural numbers smaller than 10. Then, a fitness case is one of those natural numbers. Suppose x^2+1 to be one expression to evaluate, then we say that $2^2+1=5$ is the value assumed by this expression over the fitness case

2. Raw fitness is then defined as the sum of the distances over all fitness cases between values returned by perfect solutions and values returned by individuals to be evaluated. Formally, the raw fitness f_R of an individual i, calculated over a set of N fitness cases, can be defined as:

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

where $S(i,j)$ is the value returned by the evaluation of individual i over fitness case j, $C(j)$ is the correct value for fitness case j and $k \in \mathbb{N}$.

Fitness cases are typically a small sample of the entire domain space and they form the basis for generalizing the results obtained to the entire domain space. The choice of how many fitness cases to use, and which ones, is often crucial and it may depend on the available data, on the experience of the GP designer and her knowledge of the problem.

Standardized Fitness. Standardized fitness restates the raw fitness so that a lower numerical value is always a better value. For cases in which lesser values of raw fitness are better, it can happen that standardized fitness equals raw fitness. In many cases, it is convenient and desirable to make the best value of standardized fitness equal zero. If not already the case, this can be achieved by subtracting or adding a constant. If, for a particular problem, a greater value of raw fitness is better, and the maximum possible value of raw fitness f_R^{max} is known, standardized fitness f_S of an individual i can be defined as:

$$f_S(i) = f_R^{max} - f_R(i)$$

where $f_R(i)$ is the raw fitness of i.

8.1.4 Selection

As pointed out in Chapter 3, selection depends on the phenotype of the individuals, while the genetic operators depend on their genotype. Given that what makes a difference between GP and GAs is the genotype, i.e., the structures representing the individuals evolving in the population, the selection operators of GP are identical to those of GAs. In particular, GP can be implemented using fitness proportionate selection (roulette wheel), ranking selection and tournament selection. The interested reader is referred to Section 3.1 for a presentation of these selection algorithms. In GP, tournament selection is the most common choice. Numerous other selection algorithms have been developed for GP, such as tournaments that select parents based on more than one criterion [Luke and Panait, 2002] (which allows the implementation of pseudo-multiobjective evolution), and lexicase selection [Helmuth et al., 2015] (which maintains higher levels of population diversity, a desirable property for solving most problems), just to name a few.

8.1.5 Genetic Operators

Given that GP differs from GAs in the genotype of the individuals evolving in the population, new genetic operators of crossover and mutation have to be defined for GP. The standard GP crossover and mutation are presented in the continuation.

Crossover. The crossover (sexual recombination) operator creates variation in the population by producing new offspring that consist of parts taken from each parent. The two parents, which will be called T_1 and T_2, are chosen by means of a selection algorithm. Standard GP crossover [Koza, 1992] begins by independently selecting one random point in each parent (it will be called the crossover point for that parent). The crossover fragment for a particular parent is the subtree rooted at the node lying underneath the crossover point. The first 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. The second offspring is produced in a symmetric manner. Figure 8.2 shows an example of standard GP crossover.

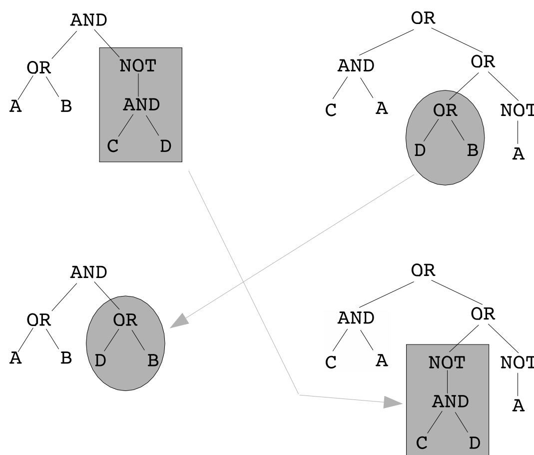

Fig. 8.2 An example of standard GP crossover. Crossover fragments are marked by gray shapes

Because entire subtrees are swapped and because of the closure property of the functions, crossover always produces syntactically legal programs, regardless of the selection of parents or crossover points.

It is important to remark that in cases where a terminal and/or the root of one parent are located at the crossover point, generated offspring could have considerable depths. This may be one possible cause of the phenomenon of *bloat*, i.e., progressive growth of the code size of individuals in the population, which will be discussed later. For this reason, many variants of the standard GP crossover have been proposed in the literature. The most common ones consist in assigning different probabilities of being chosen as crossover points to the various parents' nodes,

depending on the depth level at which they are situated. In particular, it is very common to assign low probability of being selected as crossover points to the root and the leaves, so as to limit the influence of degenerative phenomena like the ones described above. Another different kind of GP crossover is *one-point crossover*, introduced in [Poli and Langdon, 1998a]. It deserves to be mentioned for the importance it has had in the development of a solid GP theory (see Section 8.2).

Mutation. Mutation is asexual, i.e., it operates on only one parental program. Standard GP mutation, often called *subtree* mutation, begins by choosing a point at random, with uniform probability distribution, within the selected individual. This point is called the mutation point. Then, the subtree laying below the mutation point is removed and a new randomly generated subtree is inserted at that point. Figure 8.3 shows an example of standard GP mutation.

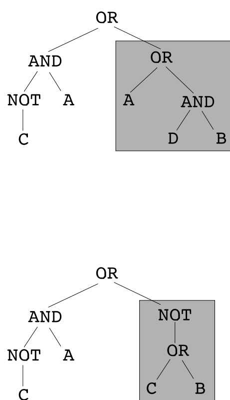

Fig. 8.3 An example of standard GP mutation

This operation, as is the case for standard crossover, is controlled by a parameter that specifies the maximum depth allowed and limits the size of the newly created subtree that is to be inserted. Nevertheless, the depth of the generated offspring can be considerably larger than that of the parent.

As for crossover, many variants of standard GP mutation have been developed too. The most commonly used are aimed at limiting the probability of selecting the root and/or the leaves of the parent as mutation points. A special mention is deserved by *point mutation* [Poli and Langdon, 1997], which exchanges a single node with a random node of the same arity, for the importance it has had in GP theory (see Section 8.2). Other commonly used variants of GP mutation are: *permutation* (or *swap* mutation), which exchanges two arguments of a node, and *shrink mutation*, which generates a new individual from a parent's subtree.

8.1.6 Survival

Once the offspring are created, the new population may completely replace the previous one, or there may be a selection of which individuals survive, among the old and new populations. Like in GAs, also in GP it is common to use some form of *elitism*, in which the best k individuals among old and new are copied unchanged into the next population, guaranteeing their survival. This parameter k is the size of the elite, and to turn off elitism one simply has to set $k = 0$. Again, like in GAs, the genetic operator of replication, described above, also represents a soft form of elitism, as it also (probabilistically) promotes the survival of the best individuals into the next population.

8.1.7 GP Parameters

Once the GP user has decided the set of functions $\mathscr{F}$ and the set of terminals $\mathscr{T}$ used to represent potential solutions of a given problem, and once the exact implementation of the genetic operators to employ has been chosen, still some parameters that characterize evolution need to be set. A list comprising some of these parameters is the following one:

- Population size.
- Population initialization algorithm.
- Selection algorithm.
- Crossover type and rate.
- Mutation type and rate.
- Maximum tree depth.
- Elitism.
- Stopping criteria.

Some of these choices arrive with more parameters to set, for example, tournament selection requires choosing the tournament size, and elitism requires choosing the elite size and the replication rate, the same way it did in GAs. The importance of parameter setting in EAs is a controversial issue. For instance, some work suggests that EAs are mostly insensitive to parameter choices [Sipper et al., 2018], while others seem to have a different opinion, at least for some versions of GP [Trujillo et al., 2020]. In general, much of what GP researchers know about parameter setting is empirical and based on experience.

Besides these parameters, an array of different implementation choices may result in wildly different evolutionary dynamics and capabilities. Like in GAs, also in GP we may have a steady-state instead of generational implementation, or a multi-objective approach, just to name a few. Specific to GP, we may have different bloat control methods, of which the maximum tree depth is one of the simplest, and we may have Automatically Defined Functions (ADFs) or other modular structures, just to name a few. The usefulness of ADFs has been shown by Koza in [Koza, 1994],

later followed by other modular constructs [Koza et al., 1999], all aimed at allowing a high degree of code reusability inside GP individuals.

8.1.8 An Example Run

A very simple example GP run is discussed here. The problem to be solved is the even parity 2 problem, and it consists in finding a Boolean function of two Boolean arguments that returns *true* if an even number of its arguments are true and *false* otherwise (a generalization called the even parity k problem will be defined later in this chapter). Let A and B be the names of the two arguments; a Boolean function $f(A,B)$ perfectly solving this problem must respect the truth table in Table 8.1. Every line of

A	B	$f(A,B)$
false	false	true
false	true	false
true	false	false
true	true	true

Table 8.1 Truth table of the optimal individual for the even parity 2 problem

this table represents a fitness case. For every Boolean function of arguments A and B, its raw fitness is defined as the number of hits over all the fitness cases. Since the maximum raw fitness value, in this case, is equal to 4, we define the standardized fitness as equal to 4 minus the raw fitness. In this way, an optimal solution has a standardized fitness equal to 0 and the worst individuals have a standardized fitness equal to 4.

Let the following set of GP parameters be used: population size of 4 individuals, tournament selection of size 2, crossover rate equal to 50%, mutation rate equal to 25% and replication rate equal to 25%, maximum tree depth equal to 4 and ramped half-and-half initialization.[4] Let individuals be built with the set of functions $\mathscr{F} = \{and, or, not\}$ and the set of terminals $\mathscr{T} = \{A, B\}$.

The process begins with the generation of the initial population by the ramped half-and-half technique. Since the maximum tree depth is 4 and the population size is 4, the initial population will probably be composed of one individual of depth 1, one individual of depth 2, one individual of depth 3 and one individual of depth 4. Let the individuals composing the initial population be the following ones (prefix expressions instead of tree representations are given for the sake of simplicity):

1. $T_1 = not(A)$
2. $T_2 = or(and(A,B), and(A,A))$
3. $T_3 = and(and(A,B), or(A,A))$

[4] The reader is warned that these parameters are unrealistic and have been used for the sake of simplicity, just to show a first example of a GP run.

4. $T_4 = not(or(B, not(or(A,B))))$

The following steps consist in evaluating the fitness of all the individuals composing the population. Truth tables of individuals T_1, T_2, T_3, T_4 are shown in Figure 8.4, tables (a) through (d), respectively. Let f_R be the raw fitness and f_S be the standardized

A	B	T_1
false	false	true
false	true	true
true	false	false
true	true	false

(a)

A	B	T_2
false	false	false
false	true	false
true	false	true
true	true	true

(b)

A	B	T_3
false	false	false
false	true	false
true	false	false
true	true	true

(c)

A	B	T_4
false	false	false
false	true	false
true	false	true
true	true	false

(d)

Fig. 8.4 (a) Truth table of individual $not(A)$; (b) Truth table of individual $or(and(A,B), and(A,A))$; (c) Truth table of individual $and(and(A,B), or(A,A))$; (d) Truth table of individual $not(or(B, not(or(A,B))))$

fitness. From these tables, the following fitness values can be calculated:

1. $f_R(T_1) = 2 \rightarrow f_S(T_1) = 2$
2. $f_R(T_2) = 2 \rightarrow f_S(T_2) = 2$
3. $f_R(T_3) = 3 \rightarrow f_S(T_3) = 1$
4. $f_R(T_4) = 1 \rightarrow f_S(T_4) = 3$

Given the rates of replication, crossover and mutation, one individual will probably be selected for replication, two individuals for crossover and one for mutation. Let the tournament selection be first applied between T_1 and T_4. T_1 has a better fitness value and thus T_1 is reproduced, i.e., copied as it is into the new population. Analogously, let the two individuals chosen with the tournament technique for crossover be T_2 and T_3 and let

$$T_5 = and(and(A,B), and(A,A)), \quad T_6 = or(and(A,B), or(A,A))$$

be the two offspring to be inserted into the new population. Finally, let T_2 be the individual selected for mutation and let:

$$T_7 = or(and(A,B), and(A, not(B)))$$

be the generated offspring. Then, the new population is:

1. $T_1 = not(A)$

2. $T_5 = and(and(A,B), and(A,A))$
3. $T_6 = or(and(A,B), or(A,A))$
4. $T_7 = or(and(A,B), and(A, not(B)))$

At this point, the process is iterated on this new population. Truth tables analogous to the ones of Figure 8.4 could be written for T_1, T_5, T_6 and T_7. These tables would allow one to calculate the following values of standardized fitness:

1. $f_S(T_1) = 2$
2. $f_S(T_5) = 1$
3. $f_S(T_6) = 2$
4. $f_S(T_7) = 2$

Let T_6 be the individual selected for replication. Let T_1 and T_7 be the individuals selected for crossover and let

$$T_8 = A, \quad T_9 = or(and(A,B), and(not(A), not(B)))$$

be the offspring to be inserted into the new population. Finally, let T_5 be the individual selected for mutation and let

$$T_{10} = and(and(A,B), B)$$

be the generated offspring. Then, the new population is:

1. $T_6 = or(and(A,B), or(A,A))$
2. $T_8 = A$
3. $T_9 = or(and(A,B), and(not(A), not(B)))$
4. $T_{10} = and(and(A,B), B)$

Iterating the process, and thus evaluating the fitness of all the individuals in the new population, T_9 is discovered to have a standardized fitness equal to 0, and thus it is an optimal solution. This leads the process to stop and to return T_9 as a solution to the given problem.

An important remark can be made on this example: consider, for instance, individual T_6: $or(and(A,B), or(A,A))$. This expression is clearly equivalent to $or(and(A,B), A)$ (in the sense that these two expressions have identical truth values). From this consideration, it is straightforward to deduce that GP individuals are not necessarily (and in general *are not*) in their most natural form. If the final user needs to have solutions in such a form, a *simplification* phase is necessary. The steps that enable simplification of the structure of solutions are often called *rewrite steps*. This phase clearly depends on the type of language used to code GP individuals and it cannot be generalized.

8.2 GP Theory

After reading the introduction to GP given in Section 8.1, one question may naturally arise: why should GP work at all? This question can be made more precise by splitting it into the following ones: why should the iterative GP process allow the building of solutions of better and better fitness? And why should it find a solution that is satisfactory for a given problem? Or even better: what is the probability of improving the fitness of solutions through the GP generations? What is the probability of finding a satisfactory solution for a given problem? The attempt to answer these questions has been an important research challenge in the GP field since its early years. The discussion contained in this section is inspired by the one in [Langdon and Poli, 2002], where Langdon and Poli offer a complete and detailed discussion of GP theory.

Being able to answer the above questions surely implies a deep understanding of what happens inside a GP population through the generations. One may think of somehow visualizing a population on a Cartesian plane. In this way, one would often find that initially the population looks like a cloud of randomly scattered points but that, through the generations, this cloud "moves" in the search space, following a well-defined trajectory. This representation would probably provide interesting information about the dynamics regulating the GP process. But, since GP is a stochastic technique, in different runs different trajectories would be observed. Moreover, it is normally impossible to visualize a search space, given its high dimensionality and complexity. Thus, one may think of recording some numerical values concerning individuals of a population through the generations and of calculating statistics on them. These numerical values may be the average fitness of the individuals, the length of the individuals, differences between parent and offspring fitness and so on. Nevertheless, given the complexity of a GP system and its numerous degrees of freedom, any number of these statistical descriptors would be able to capture only a tiny fraction of the system's characteristics.

For these reasons, the only way to understand the behavior of a GP system appears to be the definition of precise mathematical models. Theoretical studies on GAs have already been discussed in Section 3.4, where the GAs Schema Theorem was presented. Those studies have inspired the formulation of a theory for GP. This theory, consisting of a rigorous probabilistic model for GP systems, can be considered as the foundation of GP theory. This section is not intended to explain this complex subject in detail, but just to give a simple introduction.

Early GP Schema Theorems. Schema theory for GP had a slow start, one of the difficulties being that the variable-size tree structure makes it harder to develop a definition of schema. Koza [Koza, 1992] was the first one to address schema theory in GP. However, his arguments are informal and only hint at the existence of building blocks in GP. The first mathematical formulation of a schema theorem for GP is due to Altenberg [Altenberg, 1994a]. He defined a schema as a subexpression and supplied equations for the proportion of a certain individual in the population at a

certain generation and the probability that crossover picks up a certain expression from a randomly chosen program at a certain generation.

A formalization of a schema theorem for GP more similar to Holland's one for GAs is due to O'Reilly [O'Reilly and Oppacher, 1995]. Similarly to what happens for GAs, she defines a schema using a "don't care symbol" that allows us to define an order and a length for schemata. She estimates the probability of disruption of schemata by the maximum probability of disruption $P_d(H,t)$, producing the following schema theorem:

$$E[m(H,t+1)] \geq m(H,t) \frac{f(H,t)}{\overline{f}(t)} (1 - p_c P_d(H,t))$$

where $f(H,t)$ is the mean fitness of all instances of schema H and $\overline{f}(t)$ is the average fitness in the population at generation t. The disadvantage of using the maximum probability is that it may produce a very conservative measure of the number of schemata at a given generation. Moreover, the maximum probability of disruption varies with the size of a given schema and makes it very difficult to predict which schemata will tend to multiply in the population and why.

Other interesting schema theorem formulations for GP are due to Whigham [Whigham, 1996], who produced a schema theorem for a GP system based on context-free grammars, Rosca [Rosca, 1997] and Poli and Langdon [Poli and Langdon, 1998b], who gave a pessimistic lower bound (approximation) on the expected number of copies of a schema in the next generation. Next, schema theorems that give an exact expected number, rather than a bound, will be considered.

Exact GP Schema Theorem. The development of an exact and general schema theory for GP is due to Poli and colleagues [Langdon and Poli, 2002]. Syntactically, a schema is a tree with some "don't care" nodes (labelled with the symbol "=") which represent exactly one node (primitive function or terminal). Semantically, a schema represents all programs that match its size, shape and defining nodes (i.e., nodes that are not "don't care").

In addition to the definition of schema, exact GP schema theory is based on the concept of *hyperschema* and *variable-arity hyperschema* (VA hyperschema). A hyperschema is a tree composed of internal nodes from the set $\mathscr{F} \cup \{=\}$ and leaves from $\mathscr{T} \cup \{=,\#\}$, where the "#" symbol stands for any valid subtree. A VA hyperschema is a tree composed of internal nodes from the set $\mathscr{F} \cup \{=,\#\}$ and leaves from the set $\mathscr{T} \cup \{=,\#\}$, where the "=" "don't care" symbol stands for exactly one node, the terminal "#" stands for any valid subtree, while the function "#" stands for exactly one function of arity not smaller than the number of subtrees connected to it. If a function symbol "#' is matched by a function of greater arity than the number of subtrees connected to it, then some arguments of that function are left unspecified.

Many formulations of exact schema theorems for GP have been developed, depending on the type of genetic operators used (one-point crossover, point mutation, standard crossover, etc.), and on whether each member of the population (*microscopic* schema theorems) or average population properties (*macroscopic* schema

theorems) have to be considered. One of the most general forms of exact schema theorem for GP is probably the following.

Theorem 8.1 (Macroscopic Exact GP Schema Theorem for Standard Crossover).

Let selection and standard crossover be the genetic operators used by a GP system and let:

- $\alpha(H,t)$ *be the probability that the individuals produced by selection and crossover at generation t match schema H (also called total schema transmission probability for schema H).*
- p_{xo} *be the crossover rate.*
- $p(H,t)$ *be the probability of selecting an individual matching schema H to be a parent at generation t.*
- $G_1, G_2, \ldots$ *be all the possible program shapes (i.e., all the schemata of fixed size and shape, including only "=" symbols).*
- $N(K)$ *be the number of nodes in schema K.*
- $U(H,i)$ *be the hyperschema representing all the trees that match the portion of schema H above crossover point i; in other words* $U(H,i)$ *is the hyperschema obtained by replacing the subtree below crossover point i with a "#" node in H.*
- $L(H,i,j)$ *be the VA hyperschema representing all the trees that match the portion of schema H below crossover point i, but where the matching portion is rooted at some arbitrary node j.*

Then the following equality holds:

$$\alpha(H,t) = (1 - p_{xo})\, p(H,t) + p_{xo} \sum_{k,l} \frac{1}{N(G_k)N(G_l)} \sum_{i \in H \cap G_k} \sum_{j \in G_l} p(U(H,i) \cap G_k, t)\, p(L(H,i,j) \cap G_l, t) \tag{8.1}$$

An even more general formulation of the exact schema theorem exists. It holds for *any* type of subtree-swapping crossover and so it applies, for instance, to Koza's crossover, to one-point crossover and many others. The interested reader can find this formulation in [Poli and McPhee, 2003a, Poli and McPhee, 2003b]. The contribution of GP exact schema theory to the comprehension of GP systems dynamics is undeniable. For instance, in [Langdon and Poli, 2002], Poli and Langdon have performed some experimental analysis based on GP exact schema theory, which has helped in deeply understanding how GP works on some concrete problems.

8.3 GP Benchmarks

GP has been applied to many fields in industry and science and has produced a large amount of results. The attempt to classify all the applications in which GP has been used since its early beginnings is probably hopeless, even though important efforts can be found in [Langdon, 1996, Banzhaf et al., 1998]. In [Koza and Poli, 2003], Koza and Poli state that GP may be especially productive in areas having, among others, some or all of the following characteristics:

- where there is poor understanding of the problem at hand,
- where finding the size and shape of the ultimate solution is a major part of the problem,
- where there are good simulators to test performance of candidate solutions but poor methods to directly obtain good solutions,
- where conventional mathematical analysis does not, or cannot, provide analytic solutions,
- where an approximate solution is acceptable.

One more characteristic can be added here:

- where it is impossible, or difficult, to write an algorithm to solve the problem.

Problems with one or more of these characteristics can, in some sense, be called "typical GP problems". In [Koza, 1992], Koza defines a set of problems that can be considered as belonging to this class and which have the relevant property of being simple to define and suitable for application of GP. For this reason, they have been adopted by the GP research community as a, more or less, agreed upon set of benchmarks. They are introduced in Sections 8.3.1, 8.3.2 and 8.3.3. Of course this list is not exhaustive, and many other benchmarks exist and are used in GP research, but it can be considered a good set of problems to be used by researchers to test their hypotheses, since it is composed of problems of different natures and often showing different behaviors. For a more complete list of GP benchmarks that also contains pointers to freely available data, the reader is referred to [McDermott et al., 2012]. In particular, the first benchmark discussed here (symbolic regression) is a mathematical problem, the second one (even parity k) is a Boolean problem, and the third one (the artificial ant) is a simple application of path planning in artificial intelligence.

8.3.1 Symbolic Regression

We have already introduced symbolic regression in Chapter 5, since it is one of the main types of problem of Machine Learning. However, given the importance that it has in GP, it is appropriate to rediscuss here some of the main characteristics of this type of problem, and present it in the new perspective of GP.

Given a set of vectors $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n\}$, where for all $i = 1, 2, ..., n, \mathbf{x}_i \in \mathbb{R}^m$, and a vector $\mathbf{t} = [t_1, t_2, ..., t_n]$, where for all $i = 1, 2, ..., n, t_i \in \mathbb{R}$, a symbolic regression problem can be generally defined as the problem of finding, or approximating, a function $\phi : \mathbb{R}^m \to \mathbb{R}$, also called the target function, such that:

$$\forall i = 1, 2, ..., n : \ \phi(\mathbf{x}_i) = t_i$$

GP is typically used to solve symbolic regression problems using a set of primitive functions $\mathscr{F}$ that are mathematical functions, for instance the arithmetic functions or others, and a set of terminal symbols $\mathscr{T}$ that contain at least m different real-valued variables, and may also contain any set of numeric constants. In this way, a GP individual (or program) P can be seen as a function that, for each input vector $\mathbf{x}_i$, returns the scalar value $P(\mathbf{x}_i)$. In symbolic regression, to measure the fitness of an individual P any distance metric (or error) between the vector $[P(\mathbf{x}_1), P(\mathbf{x}_2), ..., P(\mathbf{x}_n)]$ and the vector $[t_1, t_2, ..., t_n]$ can be used. As an example, one may use the mean Euclidean distance, or root mean square error, and the fitness $f(P)$ of a GP individual P would be:

$$f(P) = \sqrt{\frac{\sum_{i=1}^{n} (P(\mathbf{x}_i) - t_i)^2}{n}} \tag{8.2}$$

or one may use the Manhattan distance, or absolute error, and in this case the fitness $f(P)$ of a GP individual P is:

$$f(P) = \sum_{i=1}^{n} |P(\mathbf{x}_i) - t_i| \tag{8.3}$$

Using any error measure as fitness, a symbolic regression problem can be seen as a minimization problem, where the optimal fitness value is equal to zero (in fact, any individual with an error equal to zero behaves on the input data exactly like the target function ϕ). As is customary in Machine Learning, the vectors $\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n$ are usually called input data, input vectors, training instances or *fitness cases*, while the values $t_1, t_2, ..., t_n$ are usually identified as the corresponding *target values*, or expected output values. $\mathbf{X}$ is usually called the *dataset*. Finally, the values $P(\mathbf{x}_1), P(\mathbf{x}_2), ..., P(\mathbf{x}_n)$ are usually called the *output values* of individual P on the input data.

Example 8.2. Let $\mathbf{X} = \{[3, 12, 1], [5, 4, 2]\}$ and let $\overrightarrow{t} = [27, 13]$. For instance, GP individuals may be coded using as primitive functions the set of arithmetic operators $\mathscr{F} = \{+, -, *\}$ and as terminals a set of three real-valued variables (since the input vectors have size equal to 3) $\mathscr{T} = \{k_1, k_2, k_3\}$. In this way, the search space contains all the trees that can be built by composing the symbols in $\mathscr{F}$ and $\mathscr{T}$ (with the only exception that usually a maximum possible depth is imposed on the trees, as previously discussed). Using, for instance, the absolute error, one may calculate the fitness of an individual, for instance:

$$P(k1,k2,k3) = k3 * (k1 - k2)$$

To do that, one has to first calculate the output values of P on the input data. In other words, one has to calculate $P(3,12,1)$ (obtained by substituting the values of the first input vector in the dataset $\mathbf{X}$ for k_1, k_2 and k_3 respectively in the expression of P) and $P(5,4,2)$ (obtained by substituting the values of the second input vector in the dataset $\mathbf{X}$ for k_1, k_2 and k_3 in P). So, the fitness of P is:

$$\begin{aligned} f(P) &= |P(3,12,1) - 27| + |P(5,4,2) - 13| \\ &= |(1 * (12 - 3)) - 27| + |(2 * (5 - 4)) - 13| \\ &= |9 - 27| + |2 - 13| = 18 + 11 = 29 \end{aligned}$$

It is not difficult to realize that, in this example, a global optimum, i.e., an individual that has a fitness equal to zero, is:

$$P_{opt}(k_1,k_2,k_3) = k_1 + 2 * k_2$$

From this, we can see that GP individuals do not have to necessarily use all the variables in $\mathscr{T}$ (for instance, P_{opt} does not use k_3).

In this simple example, only the binary operators of sum, subtraction and multiplication have been used. When division is also used, it is typical to "protect" it in some way from failure in case the denominator is equal to zero. The oldest and most popular method to protect division is to replace it with an operator that is equal to division if the denominator is different from zero and that returns a constant value otherwise [Koza, 1992]. Nevertheless, several more sophisticated methods have been introduced [Keijzer, 2003].

Of course, one of the main requirements for symbolic regression problems is that the evolved models have a good generalization ability (as discussed in Chapter 5).

Symbolic Regression with Synthetic Functions. Symbolic regression has a huge number of real-life applications. Nevertheless, real-world datasets are often very complex, and they may be noisy and contain errors. For this reason, when we want to study the properties of GP and other Machine Learning algorithms, it is often useful to create artificial, or synthetic, datasets with some known properties. A popular way of doing this is to consider a given known target function, and create a dataset that contains some inputs and some corresponding outputs generated by that function. The objective is to test whether GP, and/or other algorithms, are able to find that function, using only the considered points. For instance, one may consider a function (also called a quartic polynomial) like: $f(x) = x^4 + x^3 + x^2 + x$. A dataset could be created by using as input data a set of values for x (say, for instance, the first 100 natural numbers), and as corresponding targets the corresponding values of $f(x)$.

This quartic polynomial is known to be a rather simple function for GP to find. It can be made harder by defining numerical coefficients for the different terms. A

large set of other typical synthetic symbolic regression benchmarks can be found in [McDermott et al., 2012].

8.3.2 Boolean Problems

Boolean problems are similar to symbolic regression, with the only difference that both input data and the corresponding targets are Boolean values. Many real-life applications of Boolean problems exist, the most typical ones probably being the automatic synthesis of integrated electric circuits, on which GP reported several practical successes so far [Koza, 1992]. As for symbolic regression, also for Boolean problems it is possible that real-life datasets are too complex to be used when the properties of algorithms need to be investigated. For this reason, it is typical to create artificial, or synthetic, Boolean problems, with known properties, and use them as benchmarks to test the dynamics of GP and other algorithms.

One of the most popular Boolean synthetic problems is the even parity k problem [Koza, 1992]. The goal of this problem is to find a Boolean function of k Boolean arguments that returns *true* if an even number of its Boolean arguments evaluates to true, and that returns *false* otherwise. A very simple case, even parity 2, has already been introduced in Section 8.1.8, where a function perfectly solving this problem has been found. This function is:

$$f(A,B) = or(and(A,B), and(not(A), not(B)))$$

and its truth values for all possible values of its Boolean arguments A and B are represented in Table 8.1. As this table shows, each time an even number (0 or 2, in this case) of arguments is true, f returns true and each time an odd number (1, in this case) of arguments is true, f returns false. In general, the number of fitness cases, i.e., the number of all possible permutations of the values of the k parameters, is 2^k. Fitness is usually computed as 2^k minus the number of hits over the 2^k cases. Thus, a perfect individual has fitness equal to 0, while the worst individuals have fitness equal to 2^k. Unlike what happens in Section 8.1.8, a typical set of functions employed for this problem is $\mathscr{F} = \{nand, nor\}$, while the terminal set is usually composed of k different Boolean variables.

Besides the even parity k problems, other known Boolean GP benchmarks are the h-multiplexer [Koza, 1992] and the FPGA [Vanneschi, 2004].

8.3.3 The Artificial Ant on the Santa Fe Trail

In this problem [Koza, 1992], an artificial ant is placed on a 32×32 toroidal grid. Some of the cells from the grid contain food pellets. The goal is to find a navigation strategy for the ant that maximizes its food intake. The ant starts in the upper left

cell of the grid, identified by the coordinates (0, 0), facing east. It has a very limited view of its world. In particular, it has a sensor that can see only a single immediately adjacent cell in the direction the ant is currently facing. Food is placed on the grid according to the “Santa Fe trail”, an irregular trail consisting of 89 food pellets. The trail is not straight and continuous, but instead has single gaps, double gaps and triple gaps at corners. Figure 8.5 shows the Santa Fe trail. Food is represented by solid black squares, while gaps are represented by gray squares. Numbers identify key characteristics along the trail, for example the number 3 highlights the first corner, number 11 the first single gap, and so on.

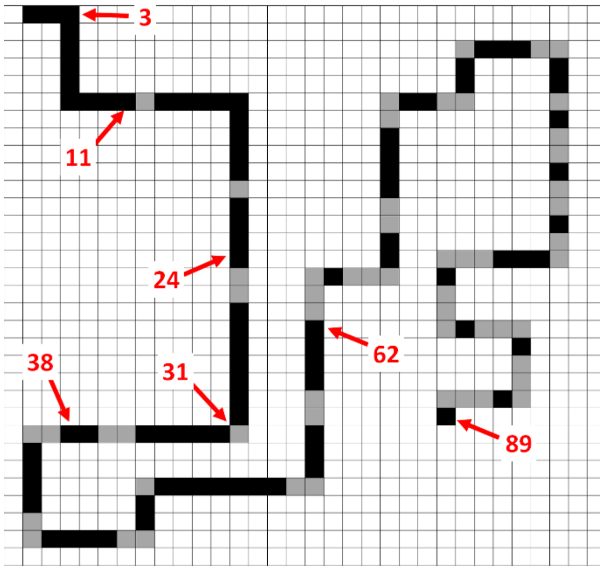

Fig. 8.5 The Santa Fe trail used for the artificial ant problem

The set of terminals used for this problem is usually $\mathscr{T} = \{Right, Left, Move\}$, and corresponds to the actions the ant can perform: rotate to the right by 90°, rotate to the left by 90°, and move forward in the currently facing direction. When the ant moves into a square containing a food pellet, it eats the food (thereby eliminating the pellet from that square and erasing the trail). The set of primitive functions that are typically used for this problem is $\mathscr{F} = \{IfFoodAhead, Progn2, Progn3\}$. $IfFoodAhead$ is a conditional branching operator that takes two arguments and executes the first one if and only if a food pellet is present in the case that it is adjacent to the ant in the direction the ant is facing, and the second one otherwise (it represents the information the ant can get by its sensor). *Progn*2 and *Progn*3 are common LISP operators. *Progn*2 takes two arguments and causes the ant to unconditionally execute the first argument, followed by the second one; *Progn*3 is analogous, but it takes three arguments, which are executed in an ordered sequence. An individual built with these sets $\mathscr{F}$ and $\mathscr{T}$ can be considered as a “program” that allows the ant

to navigate the grid. As fitness function, the total number of food pellets lying on the trail (89) minus the amount of food eaten by the ant in a fixed amount of time is considered. This turns the problem into a minimization one, like the previous ones. Each move operation and each rotate operation is considered to take one time step, and the ant is usually limited to 600 time steps. This timeout limit is sufficiently small to prevent a random walk of the ant to cover all the 1024 squares, and thus eat all the food, before timing out.

8.4 GP Open Issues

8.4.1 GP Problem Difficulty

The capability of GP to find good solutions varies from problem to problem. To convince oneself of this fact, one could perform the following experiment: execute the artificial ant on the Santa Fe trail problem (see Section 8.3.3) for, say, n generations; record the values $f_1, f_2, \ldots, f_n$, where, for each $g = 1, 2, ..., n$, f_g is the fitness of the best individual found in the population at generation g. Since GP is based on stochastic operators, these values may be fortuitous and surely lack statistical significance. For this reason, as we have studied for GAs in Section 3.6, it is appropriate to repeat the execution a number of times, say 100, and consider average (or median) results. Thus, let $\overline{f}_g$ be the average of the best fitness values found at generation g over the 100 runs performed and plot on a Cartesian two-dimensional plane the values $\overline{f}_1, \overline{f}_2, \ldots, \overline{f}_n$ (on the ordinates) against generation numbers (on the abscissas). Now, repeat the same process for a symbolic regression problem (see Section 8.3.1), aimed at finding a simple equation, like for instance $x^4 + x^3 + x^2 + x$ (a quartic polynomial). Let the GP parameters used to solve these two different problems be identical (except, of course, for the sets of functions $\mathscr{F}$ and of terminal symbols $\mathscr{T}$). Figure 8.6 shows the results of this experiment, where the set of GP parameters for both problems was the following one: population size of 2500 individuals, ramped half-and-half initialization, tournament selection of size 10, crossover rate equal to 95% (standard GP crossover [Koza, 1992]), mutation rate equal to 0.1% (standard subtree mutation [Koza, 1992]), maximum tree depth for the initialization phase equal to 6, maximum depth for the crossover and mutation phases equal to 17, elitism (with elite size equal to 1). In the case of the symbolic regression problem, 100 equidistant points included in the range $[0, 5]$ have been used as fitness cases. The blue curve represents results of artificial ant and the black one results of symbolic regression. This figure clearly shows that, for the symbolic regression problem, the optimal solution (fitness equal to zero) has been found (or at least appropriately approximated) in all the executed runs before generation 8. This is surely not the case for the artificial ant problem, since the blue curve does not touch the abscissas axis during the first 40 generations. Data not shown here confirm that, even if generations until 500 were considered, still the blue curve would

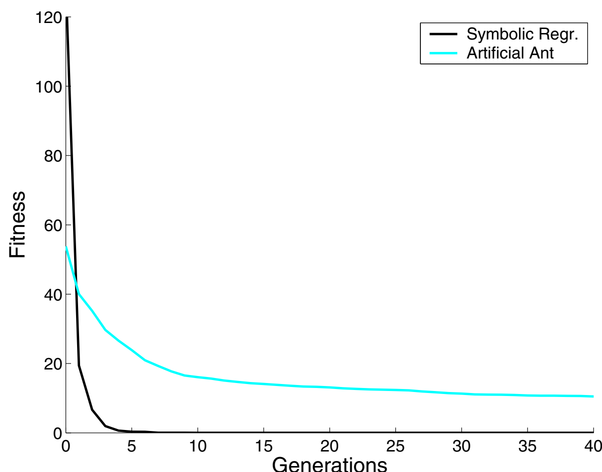

Fig. 8.6 Symbolic regression problem and artificial ant problem. Evolution of best fitness during the first 40 generations. The two curves have been obtained using the same GP parameters (see text) and are averages over 100 independent GP runs

not touch the abscissas axis. Moreover, fewer than 50 runs, over the total 100 runs executed, have led to the discovery of one optimal solution before generation 500 for the artificial ant problem.

These different behaviors are usual in GP, when considering different problems. Thus, the following questions arise naturally: why does it happen? Why are some problems easily solved by GP, while others are so hard? The following observations can be made about the experiment shown in Figure 8.6:

(i) The GP parameters used for the two considered problems are exactly the same.
(ii) The range of all possible fitness values that an individual of the symbolic regression problem can take is much larger than the ones that an individual of the artificial ant can take. In fact, fitness values for the ant problem range from 0 to 89 by definition (since 89 is the number of food pellets on the Santa Fe trail), while the best fitness value in the population at generation 0 for the symbolic regression problem was, on average, around 120 in the experiment shown (see Figure 8.6). Furthermore, the "granularity" in the fitness function is very different: only 90 different fitness values are possible for the ant problem, while fitness is continuous for symbolic regression, and so an infinite number of values are possible.
(iii) Trees in the ant problem can be built using just three primitive functions (*IfFoodAhead*, *Progn2*, *Progn3*), while in the symbolic regression problem, they can be built using four primitive functions ($+$, $-$, $*$ and $//$). Furthermore, arities of the operators are also different: all the operators used for symbolic regression have an arity equal to 2, while two operators used for the ant have an arity equal to 3 and one of them has an arity equal to 2.

Observation (i) leads to the conclusion that GP parameters, even though they can have an influence on the performance of a GP system, do not give a sufficient amount of information to fully understand its dynamics and justify its behavior. Observations (ii) and (iii) allow one to conclude that the size of the set of all possible fitness values and the size of the set of all possible trees, although probably not irrelevant, are not valid criteria to establish the difficulty of a problem for GP. In other words, the structure of the fitness landscape, and not only its size, matters.

Thus, answering the question "what makes a problem easy or hard for GP?" does not seem to be an easy task. Several works were published in the last decade, trying to given an answer to this ambitious question, many of them collected in [Vanneschi, 2004]. The most relevant are arguably the ones aimed at defining mathematical measures able to capture some characteristics of fitness landscapes that can have implications for the difficulty of the problem. The most successful of these measures, even though not without flaws, is the *fitness-distance correlation* [Tomassini et al., 2005], indicating that the relationship between fitness and distance to the goal can give a reliable indication of the hardness of the problem, assuming that the distance is appropriately related to the genetic operators used to evolve [Gustafson and Vanneschi, 2005]. Another measure that can give some useful indications is the *negative slope coefficient* [Vanneschi et al., 2006], based on the concept of *fitness clouds* [Vanneschi et al., 2004].

8.4.2 Premature Convergence or Stagnation

After a certain number of generations, GP populations, as is the case for GAs and any other evolutionary algorithm, tend to contain individuals that are similar, or even identical, to each other. To show this phenomenon, measures analogous to the ones introduced in Section 3.5.1 for GAs can be used. Many reasons can be given for premature convergence. First of all, the selection mechanism acts on phenotypes, thus assigning high probabilities of surviving to only a few good-quality individuals. Secondly, according to schema theory, building blocks tend to multiply in the population, thus introducing similar substructures inside individuals [Langdon and Poli, 2002].

Many techniques have been developed by GP researchers to maintain diversity, both genotypic and phenotypic, inside GP populations (see, for instance, [Ekárt and Németh, 2002]). Although these techniques have often proved very useful to understand why premature convergence happens, their main drawback is generally that they are time consuming, since they demand the calculation of measures like structural distances, entropy or variance, at each generation. This generally causes an increment in the total completion time of GP systems, which may make them unusable in practice (even though efficient methods to approximate diversity measures have been defined [Burke et al., 2002]). Otherwise, new genetic operators, like different kinds of selections, crossovers or mutations, have been defined in order to produce offspring that are as different as possible from their par-

ents, or from the other individuals in the population, by construction. These techniques generally succeed in maintaining diversity inside populations, but substantially change the GP algorithm (in particular the genetic operators used) and thus the behavior of GP systems. [Fernández et al., 2003] demonstrated that parallelizing the GP process, and organizing it into separate and interacting subpopulations, helps to limit premature convergence and its deleterious effects on GP. The migration of candidate solutions from one population to another, in fact, allows individuals having different ancestors, and thus possibly different evolutionary histories and behaviors, to enter converging populations and to inject diversity inside them. Moreover, since (copies of) the fitter individuals of each subpopulation are sent to the other ones, good quality of solutions should be maintained together with diversity (one could informally say that not just "diversity", but "good diversity" is maintained).

8.4.3 The Problem of Bloat

The search space of GP is virtually unlimited and programs tend to grow in size during the evolutionary process. Code growth is a healthy result of genetic operators in search of better solutions, but it also permits the appearance of pieces of redundant code that increase the size of programs without improving their fitness. These "useless" pieces of code are often called *introns*, based on the name given to their biological counterparts. Introns can be roughly divided into two categories: inviable code and unoptimized code (or syntactic/structural and semantic introns [Angeline, 1998, Brameier and Banzhaf, 2003]). The former is code that cannot contribute to the fitness no matter how many changes it suffers, either because it is never executed or because its return value is ignored. The latter is viable code containing redundant elements whose removal would not change the return value [Luke, 2003]. Besides consuming precious time in an already computationally intensive process, redundant code may start growing rapidly, a phenomenon known as bloat [Banzhaf et al., 1998, Chapter 7], [Langdon and Poli, 2002, Chapter 11]. Bloat can be defined as an excess of code growth without a corresponding improvement in fitness. This is a serious and widely studied problem in GP, often being a main cause of stagnation of the evolutionary process.

The different explanations for bloat given through the years are not necessarily contradictory. Some appear to be generalizations or refinements of others, and several most certainly complement each other. An extensive review of the first five theories presented below can be found in [Silva and Costa, 2009]. A detailed explanation of the last theory, Crossover Bias, can be found in [Silva et al., 2012].

Hitchhiking. One of the first explanations for the proliferation of introns among GP programs, advanced by Tackett, was the hitchhiking phenomenon [Tackett, 1994]. This is a common and undesirable occurrence in genetic algorithms where unfit building blocks propagate throughout the population simply because they happen to adjoin highly fit building blocks. According to the hitch-

hiking explanation, the reason why naturally emerging introns in GP become so abundant is that they, too, are hitchhikers.

Defense Against Crossover. The idea of defense against crossover as being the explanation for bloat has persisted in the literature for a long time [Altenberg, 1994b, Blickle and Thiele, 1994, McPhee and Miller, 1995, Nordin and Banzhaf, 1995, Smith and Harries, 1998]. It is based on the fact that standard crossover is usually very destructive [Banzhaf et al., 1998, Chapter 6], [Nordin and Banzhaf, 1995, Nordin et al., 1996]. In face of a genetic operator that seldom creates offspring better than their parents, particularly in more advanced stages of the evolution, the advantage belongs to individuals that at least have the same fitness as their parents, i.e., those who were created by neutral variations. Introns provide standard crossover and other genetic operators with genetic material where swapping can be performed without harming the effective code, leading to these neutral variations.

Removal Bias. Although supporting the defense against crossover theory, Soule performed additional experiments and concluded that there must be a second cause for code growth, presenting a theory called removal bias [Langdon et al., 1999, Soule and Foster, 1998]. The presence of inviable code provides regions where removal or addition of genetic material does not modify the fitness of the individual. According to removal bias, to maintain fitness the removed branches must be contained within the inviable region, meaning they cannot be deeper than the inviable subtree. On the other hand, the addition of a branch inside an inviable region cannot affect fitness regardless of how deep the new branch is. This asymmetry can explain code growth, even in the absence of destructive genetic operators.

Fitness Causes Bloat. The first theory that does not make introns responsible for bloat was advanced by Langdon and Poli [Langdon, 1998, Langdon and Poli, 1997, Langdon and Poli, 1998, Langdon et al., 1999]. The fitness causes bloat theory basically states that with a representation of variable length there are many different ways to represent the same program, long and short, and a static evaluation function will attribute the same fitness to all, as long as their behavior is the same. Given the inherent destructiveness of crossover, when better solutions become hard to find there is a selection bias towards programs that have the same fitness as their parents. Because there are many more longer ways to represent a program than shorter ways, a natural drift towards longer solutions occurs, causing bloat.

Modification Point Depth. Another explanation for bloat was advanced by Luke and is usually referred to as modification point depth [Luke, 2003]. Confirming previous results [Igel and Chellapilla, 1999], Luke observed that when a genetic operator modifies a parent to create an offspring, the deeper the modification point, the smaller the change in fitness. Once again because of the destructive nature of crossover, small changes will eventually benefit from a selective advantage over large changes, so there is a preference for deeper modification points, and consequently larger trees. Plus, the deeper the modification point, the smaller the branch that is removed, thus creating a removal bias [Soule and Heckendorn, 2002].

Crossover Bias. Dignum and Poli [Dignum and Poli, 2007, Dignum and Poli, 2008, Poli et al., 2007, Poli et al., 2008b] explain code growth in

tree-based GP by the effect that standard subtree crossover has on the distribution of tree sizes in the population. The average length of programs in the mating pool does not differ from that of the resultant child population after the application of crossover. However, the length distribution of the child population, under normal GP experimental conditions, is biased towards smaller programs. Smaller programs will be unable to obtain reasonable fitness and will be discarded by selection, hence increasing the average size of programs in the mating pool for the succeeding generation.

Regarding methods for counteracting bloat, they are very numerous, and act at different phases of the evolutionary cycle. For example, when evaluating fitness, the *parametric parsimony pressure* method, e.g., [Zhang and Mühlenbein, 1995] adds a term to the fitness calculation that penalizes larger individuals, while the *Tarpeian* method [Poli, 2003] periodically "kills" a fraction of individuals with above-average size by giving them an extremely bad fitness value; when selecting the parents for breeding, size may also be taken into consideration, either in a true multiobjective setting, e.g., [Bleuler et al., 2001] or using modified size-aware tournaments [Luke and Panait, 2002]; different types of genetic operators have also been developed that counteract bloat, such as *homologous crossover* [Langdon, 1999]; regarding selection for survival, Koza's maximum depth is still the most common method [Koza, 1992], while others include dynamic depth and size limits [Silva and Costa, 2009] and the *operator equalisation* method [Silva and Dignum, 2009]. Other types of methods do not fit into any particular phase of the evolutionary cycle. A taxonomy and survey of most bloat control methods can be found in [Silva et al., 2012].

8.5 Geometric Semantic Genetic Programming

8.5.1 Semantics in Genetic Programming

Geometric Semantic GP (GSGP) is the name we give to GP that uses two novel genetic operators, called geometric semantic operators, instead of the traditional crossover and mutation. These new operators are based on the concept of semantics that is briefly introduced here. As discussed previously, let $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n\}$ be the set of input data, or fitness cases, of a supervised learning problem, and $\mathbf{t} = [t_1, t_2, ..., t_n]$ the vector of the respective expected output or target values. Let P be a GP individual (or program). As discussed previously, P can be seen as a function that, for each input vector $\mathbf{x}_i$, returns a value $P(\mathbf{x}_i)$. Following [Moraglio et al., 2012], the *semantics* of P is given by the vector:

$$\mathbf{s_P} = [P(\mathbf{x}_1), P(\mathbf{x}_2), ..., P(\mathbf{x}_n)]$$

In other words, from now on, we indicate with the term semantics the vector of the output values of a GP individual on the input data. Even though the concept of

semantics is general, and applies to any type of supervised learning problem, it is particularly intuitive to discuss the case of symbolic regression. So, for simplicity, and without forgetting that it is just a particular case study while the concepts are general, let us assume that the problem at hand is a symbolic regression one. In this case, $\mathbf{s_P} \in \mathbb{R}^m$, and so $\mathbf{s_P}$ can be represented as a point in an n-dimensional Cartesian space, which we call *semantic space*. Remark that the target vector $\mathbf{t}$ itself is a point in the semantic space and, since we are dealing with supervised learning, its exact location is known. What is not known is the tree structure of a GP individual that has $\mathbf{t}$ as its own semantics. Basically, when working with GP, one may imagine the existence of two different spaces: one that we call genotypic or syntactic space, in which GP individuals are represented by tree structures (or, according to the different type of GP, linear structures, graphs, etc.), and one that we call semantic space, in which GP individuals are represented by their semantics, and thus by points. The situation is exemplified in Figure 8.7, where, for simplicity, the semantic space is represented in two dimensions, which corresponds to the toy case in which only two training instances exist. The figure also shows that to each tree

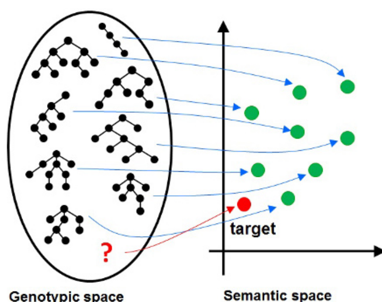

Fig. 8.7 When we work with GP, we can imagine the existence of two spaces: the genotypic space, in which individuals are represented by their structures, and the semantic space, in which individuals are represented by points, which are their semantics. In the figure, the semantic space is represented in 2D, which corresponds to the unrealistic case in which only two training instances exist

in the genotypic space there corresponds a point in the semantic space. This correspondence is surjective and, in general, not bijective, since different trees can have the same semantics. Once we have the semantics of a GP individual, its fitness can be calculated as the distance between the semantics and the target, using any metric. For instance, Equation (8.2) on page 222 is the mean Euclidean distance between $\mathbf{s_P} = [P(\mathbf{x}_1), P(\mathbf{x}_2), ..., P(\mathbf{x}_n)]$ and $\mathbf{t} = [t_1, t_2, ..., t_n]$, while Equation (8.3) on page 222 is the Manhattan distance between $\mathbf{s_P}$ and $\mathbf{t}$.

Example 8.3. Let us consider the same symbolic regression problem as in Example 8.2 (page 222). In that problem, the set of input data was $\mathbf{X} = \{[3,12,1],[5,4,2]\}$ and the vector of the corresponding target values was $\mathbf{t} = [27,13]$. In that example, we have considered the set of primitive functions $\mathscr{F} = \{+,-,*\}$ and a set

of terminals composed of three real-valued variables $\mathscr{T} = \{k_1, k_2, k_3\}$. Also, in that example, we have studied an individual whose expression in infix notation is $P(k1, k2, k3) = k3 * (k1 - k2)$. What is the semantics of this individual?

The semantics of $P(k1, k2, k3) = k3 * (k1 - k2)$, in this case, is a vector of dimension equal to 2, because there are two training instances (the vectors contained in dataset $\mathbf{X}$), where the first component is equal to the evaluation of P on the first input vector and the second component is equal to the evaluation of P on the second input vector. In other words, the first component is obtained by substituting the values 3, 12 and 1 respectively for k_1, k_2 and k_3, and evaluating P. Analogously, the second component is obtained by substituting the values 5, 4 and 2 for k_1, k_2 and k_3. So:

$$\overrightarrow{s_P} = [P(3,12,1), P(5,4,2)] = [1*(12-3), 2*(5-4)] = [9,2]$$

Using, for instance, the absolute error between semantics and target, we have that the fitness of P is:

$$f(P) = d([9,2],[27,13]) = |9-27| + |2-13| = 18 + 11 = 29$$

where, in this case, d is the Manhattan distance. Comparing this calculation with that of Example 8.2 (page 222) to calculate the fitness of P, we can clearly see that these two calculations are completely identical.

8.5.2 Similarity Between Semantic Space of Symbolic Regression and Continuous Optimization

The reader is now invited to have a look back at Section 3.7, page 102, where continuous optimization problems were introduced, and geometric genetic operators of GAs were presented for those problems. Let us now consider a particular instance of a continuous optimization problem, with the following characteristics:

- $S = \{\mathbf{v} \in \mathbb{R}^n \mid \forall i = 1, 2, ..., n : v_i \in [\alpha_i, \beta_i]\}$
- $\forall \mathbf{v} \in S: \ f(\mathbf{v}) = d(\mathbf{v}, \mathbf{t})$
- Minimization problem.

where d is a distance measure (think, for simplicity, of the Euclidean distance, but the reasoning holds also for any other metric), and $\mathbf{t} \in \mathbb{R}^n$ is a predefined, and known, global optimum. In informal terms, solutions of this problem instance are vectors of n real numbers included in a predefined interval, and the fitness of an individual is equal to the distance of that individual to a unique, and known, globally optimal solution. Let us now assume that we are trying to solve this problem with GAs, using geometric mutation (also called box mutation, or ball mutation). As we have studied in Section 3.7, in this situation, the fitness landscape of this problem is unimodal. In other terms, there are no local optima, except for the global optimum. For this reason, the problem is characterized by a very good evolvability. This fact can be easily seen by implementing it. It will not be difficult to verify that,

simply applying geometric mutation iteratively, it is possible to find solutions that are arbitrarily close to the global optimum. Just to fix some simple terminology, let a problem with these characteristics be called *CONO*, which stands for *C*ontinuous *O*ptimization with k*N*own *O*ptimum. We find this acronym particularly appropriate because the shape of the fitness landscape for this problem is actually a cone[5], where the vertex represents the global optimum. In synthesis:

> *IF box mutation is the operator used to explore the search space*
> *THEN The CONO problem has a unimodal fitness landscape.*

Let us now observe the right part of Figure 8.7, showing the semantic space of a symbolic regression problem. In this space, individuals are points and the target is known. It should not be hard to understand that this space is identical to the space of solutions of the CONO problem. This fact has a very important consequence for the solution of symbolic regression problems using GP. Similarly to the above property, we could now write:

> *IF we define a GP operator that works like box mutation on the semantic space*
> *THEN* **any** *symbolic regression problem has a unimodal fitness landscape.*

Let us repeat this property again, but with different words:

> *If we are able to define, on the syntax of the GP individuals (i.e., on their tree structure)*[6] *an operator that has, on the semantic space, the same effect as box mutation, then the fitness landscape is unimodal for any symbolic regression problem.*

In other terms, if we were able to define such an operator, then we would be able to map any symbolic regression problem into an instance of the CONO problem that uses box mutation, in the sense that we would be able to perform exactly the same actions in a space (the semantic space of GP) that is identical. As such, we would inherit the same fitness landscape properties, i.e., a unimodal fitness landscape. It is worth stressing here one point:

> *This property would hold for any symbolic regression problem, independently of how large or complex the data of the problem are.*

Since no locally optimal solution exists (except for the global optima), actually *any* symbolic regression problem could be *easy* to optimize for GP (at least on the training set, where fitness is calculated), including problems characterized by huge amounts of data. This fact would clearly foster GP as a very promising method for facing the new challenges of *Big Data* [Fan and Bifet, 2013].

At this point, beginner readers, for instance students, are invited to stop for a second and reflect on the importance and impact that this would have. For years, one of the main justifications that researchers have given to the limitations of GP was the fact that it was extremely difficult to study its fitness landscapes, because of the

[5] The word "cono" actually means cone in several languages of Latin origin, including Italian and Spanish.

[6] How could it be otherwise? GP is working with a population of trees, so the genetic operators can only act on them!

extreme complexity of the genotype/phenotype mapping, and because of the complexity of the neighborhoods induced by GP crossover and mutation. Introducing such an operator, we would not have to worry about this anymore! For any symbolic regression problem, we would have the certainty that the fitness landscape is unimodal, and thus easily evolvable. Furthermore: how many ML methods do you know in which the error surface is guaranteed to be unimodal for any possible application? Saying that this GP system would be the first ML system to induce unimodal error surfaces would probably be inappropriate; it is not impossible that other machine learning systems with this characteristic may have been defined so far. But still, it is absolutely legitimate to say that this characteristic is quite rare in ML, and should give a clear advantage to GP, compared to other well-known systems, at least in terms of evolvability.

All we have to do to obtain such an important result is to define an operator that, acting on trees, has an effect on the semantic space that is equivalent to box mutation. In other words, if we mutate a GP individual P_1, obtaining a new individual P_2, the semantics of P_2 should be like the semantics of P_1 except for a perturbation of its coordinates, whose magnitude is included in a predefined range. This is the objective of geometric semantic mutation, which is defined in the continuation.

8.5.3 Geometric Semantic Mutation

The objective of Geometric Semantic Mutation (GSM) is to generate a transformation on the syntax of GP individuals that has the same effect on their semantics as box mutation. The situation is exemplified in Figure 8.8. More particularly, this

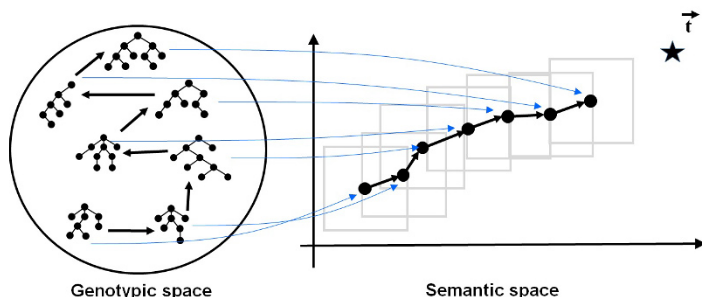

Fig. 8.8 A simple graphical representation, in the genotypic space, of a chain of solutions $\mathscr{C} = \{s_1, s_2, ..., s_h\}$ where, for each $i = 1, 2, ..., h-1$, s_{i+1} is a neighbor of s_i, and the corresponding points in the semantic space. The known vector of target values is represented in the semantic space by a star

figure represents a chain of possible individuals that could be generated by applying GSM several times, with their corresponding semantics. Given that the seman-

tics of the individuals generated by mutation can be any point inside a box of a given side centered on the semantics itself, GSM has always the possibility of creating an individual whose semantics is closer to the target (represented by a star symbol in the figure). As a direct consequence, the fitness landscape has no locally optimal solutions. Comparing the semantic space of Figure 8.8 with Figure 3.18 (page 103), it should not be difficult to see that if we use GSM, we *are* actually performing a CONO problem with box mutation on the semantic space. The definition of GSM, as given in [Moraglio et al., 2012], is as follows.

Definition 8.1. Geometric Semantic Mutation (GSM). Given a parent function $P : \mathbb{R}^n \to \mathbb{R}$, geometric semantic mutation with mutation step *ms* returns the function $P_M = P + ms \cdot (T_{R1} - T_{R2})$, where T_{R1} and T_{R2} are random functions.

It is not difficult to have an intuition of the fact that GSM has the same effect as box mutation on the semantic space. In fact, one should consider that each element of the semantic vector of P_M is a “weak” perturbation of the corresponding element in P’s semantics. We informally define this perturbation as “weak” because it is given by a random expression centered at zero (the difference between two random trees). Nevertheless, by changing parameter *ms*, we are able to tune the ”step” of the mutation, and thus the importance of this perturbation. Figure 8.9 gives a visual representation of the tree generated by GSM on a simple example. The tree P that

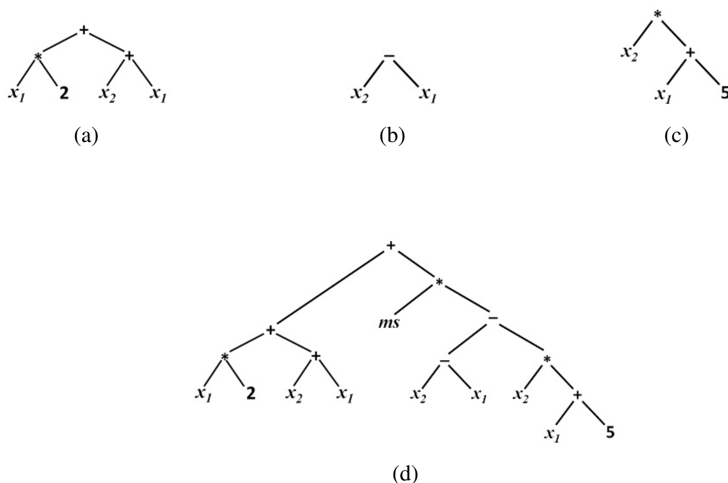

Fig. 8.9 (a) A GP individual P that is going to be mutated; (b) and (c) Two random trees T_{R1} and T_{R2}; (d) The individual P_M generated by geometric semantic mutation.

is mutated is represented in Figure 8.9a. The two used random trees T_{R1} and T_{R2} are shown in Figure 8.9b and Figure 8.9c respectively. Finally, Figure 8.9d shows the resulting tree P_M generated by GSM.

In practice, and in order to make the perturbation even weaker, it is often useful to limit the codomain of the possible outputs of T_{R1} and T_{R2} to a given predefined range. This allows us to better "control" what mutation can do. The typical situation is that T_{R1} and T_{R2} are forced to assume values in $[0,1]$. This can be done easily, for instance, by "wrapping" T_{R1} and T_{R2} inside a logistic function. In other words, random trees are generated and the logistic is applied to their output before plugging them into P_M. This practice turns out to be extremely important for overfitting control in GSGP, as we will see later.

Before continuing, it is extremely important to stop for a while and convince oneself about the importance of having random *trees/expressions* (like T_{R1} and T_{R2}) in the definition of GSM, instead of just having random constants. The motivation is the following: when we mutate an individual, we want to perturb each coordinate of its semantics by a *different* amount. It is easy to understand why this is important by considering a simple numeric example. Let the semantic of an individual P be, say, $\mathbf{s_P} = [5.6, 9.2]$, and let the target vector be $\mathbf{t} = [9, 10]$ (we are again considering the unrealistic two-dimensional case for simplicity). If we used just random numbers for applying the perturbation, then we would only be able to mutate each one of the coordinates by the same amount. So, if we mutate by, say, 0.5, we obtain a new individual P_M whose semantics is $[6.1, 9.7]$. Even though P_M has a semantics that is closer to the target than P, it should not be difficult to convince oneself that if we iterate mutation, even if we change the size of the perturbation at each step, we will never have any chance of reaching the target. The only possibility that we have to reach the target is to mutate the first coordinate *more* (i.e., by a larger amount) than the second one, simply because the first coordinate of $\mathbf{s_P}$ is further away from the corresponding coordinate of $\mathbf{t}$ than the second coordinate. So, we need a way of doing a perturbation that has to possess the following properties:

- it has to be random;
- it has to be likely to be different for each coordinate;
- it does not have to use any information from the dataset.

By the third point, we mean that the algorithm that makes the perturbation cannot have a form like:

"**if** ($\mathbf{x}_i$ = ...) **then** perturbation = ..."

because in this case the system would clearly overfit the training data: the final individual would actually have a form that is very similar to that of Equation 5.1 (page 118), which was identified as one of the most typical overfitting models, because it simply mimics the data, instead of learning its underlying model.

Under these hypotheses, the only way we could imagine of doing the perturbation was to sum to the value calculated by individual P the value calculated by a random expression. A random expression, in fact, is likely to have different output values for the different fitness cases. Last but not least, the fact that the difference between two random expressions is used ($T_{R1} - T_{R2}$) instead of just one random expression can be justified as follows. Especially in the final part of a GP run, it may happen that some of the coordinates have already been approximated in a satisfactory way,

while it is not the case for others. In such a situation, it would be useful to have the possibility of modifying *some* coordinates and *not* modifying (or, which is the same, modifying by an amount equal to zero) others. The difference between two random expressions is a random expression *centered at zero*. This means that its output value is more likely to be equal to zero, or close to zero, than to be equal to any other value. In other words, by using the difference between two random expressions, we are imposing that some coordinates may have a perturbation that is likely to be equal, or at least as close as possible, to zero.

8.5.4 Geometric Semantic Crossover

GP practitioners may be surprised to notice that, so far, basically only mutation has been considered, practically ignoring crossover. Crossover, in fact, is known to be the most powerful genetic operator, at least in standard GP. This section intends to fill this gap, by presenting a Geometric Semantic Crossover (GSC) that can behave, on the semantic space, like geometric crossover of GAs in continuous optimization, defined in Section 3.7. Geometric GAs crossover works by generating one offspring that has, for each coordinate, a linear combination of the corresponding coordinates of the parents with coefficients in $[0,1]$, whose sum is equal to 1. Under these conditions, the offspring can geometrically be represented as a point that stands in the segment joining the parents. This is the behavior that GSC must have on the semantic space, as exemplified in Figure 8.10. The objective of GSC is to generate

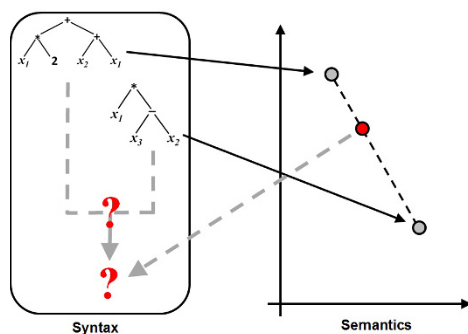

Fig. 8.10 Graphical representation of geometric semantic crossover, in the simple case of two-dimensional semantic space. The offspring generated by this crossover has a semantics that stands in the segment joining the semantics of the parents

the tree structure of an individual whose semantics stands in the segment joining the semantics of the parents. Following [Moraglio et al., 2012], GSC is defined as follows:

Definition 8.2. Geometric Semantic Crossover (GSC). Given two parent functions $T_1, T_2 : \mathbb{R}^n \rightarrow \mathbb{R}$, the geometric semantic crossover returns the function $T_{XO} = (T_1 \cdot T_R) + ((1 - T_R) \cdot T_2)$, where T_R is a random function whose output values range in the interval $[0, 1]$.

It is not difficult to see from this definition that T_{XO} has a semantics that is a linear combination of the semantics of T_1 and T_2, with random coefficients included in $[0, 1]$ and whose sum is 1. The fact that we are using a random expression T_R instead of a random number can be interpreted analogously to our explanation of the use of random expressions in GSM. Furthermore, it is worth mentioning that in Definition 8.2 the fitness function is supposed to be the Manhattan distance; if Euclidean distance is used, then T_R should be a random constant instead. The interested user is referred to [Moraglio et al., 2012] for an explanation of this concept. Figure 8.11 gives a visual representation of the tree generated by GSC on a simple example.

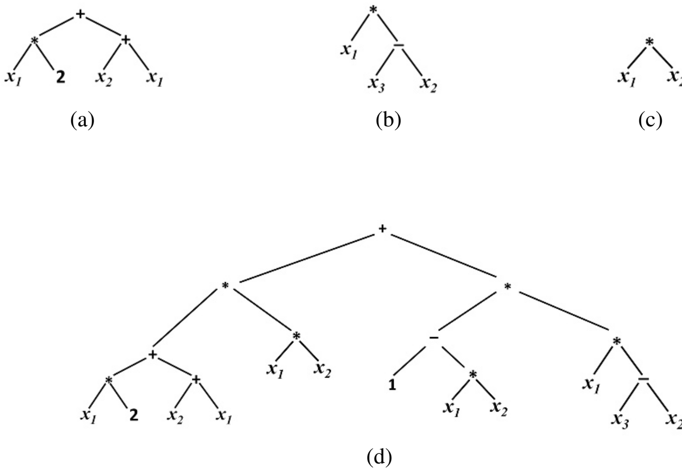

Fig. 8.11 Two parents T_1 and T_2 (plots (a) and (b) respectively), one random tree T_R (plot (c)) and the offspring T_{XO} of the crossover between T_1 and T_2 using T_R (plot (d))

The fact that the semantics of the offspring T_{XO} stands in the segment joining the semantics of the parents T_1 and T_2 has a very interesting consequence: the offspring generated by GSC cannot be worse than the worst of its parents, a property that has already been studied in Section 3.7 for GAs geometric crossover. For a deeper discussion of this property, the reader is referred to [Moraglio, 2008, Moraglio et al., 2012].

8.5.5 Code Growth and New Implementation

Looking at their definition (and at Figures 8.9 and 8.11), it is not hard to see that geometric semantic operators create offspring that contain the complete structure of the parents, plus one or more random trees and some additional arithmetic operators: the size of the offspring is thus clearly much larger than the size of their parents. The rapid growth of the individuals in the population, shown by Moraglio et al. [Moraglio et al., 2012], makes these operators unusable in practice: after a few generations the population becomes unmanageable because the fitness evaluation process becomes unbearably slow. The solution suggested in [Moraglio et al., 2012] consists in performing an automatic simplification step after each generation in which the individuals are replaced by (hopefully smaller) semantically equivalent ones. However, this additional step adds to the computational cost of GP and is only a partial solution to the progressive size growth. Last but not least, depending on the particular language used to code individuals and the used primitives, automatic simplification can be a very hard task.

In this section, we present a novel implementation of GP using these operators that overcomes this limitation, making them efficient without performing any simplification step. This implementation was first presented in [Vanneschi et al., 2013, Castelli et al., 2015]. Although the algorithm is described assuming the representation of the individuals is tree based, the implementation fits any other type of representation.

In a first step, we create an initial population of (typically random) individuals, exactly as in standard GP. We store these individuals in a table (which we call $\mathscr{P}$ from now on) as in the example reported in Figure 8.12a, and we evaluate them. To

Id	Individual
T_1	$x_1 + x_2 x_3$
T_2	$x_3 - x_2 x_4$
T_3	$x_3 + x_4 - 2x_1$
T_4	$x_1 x_3$
T_5	$x_1 - x_3$

(a)

Id	Individual
R_1	$x_1 + x_2 - 2x_4$
R_2	$x_2 - x_1$
R_3	$x_1 + x_4 - 3x_3$
R_4	$x_2 - x_3 - x_4$
R_5	$2x_1$

(b)

Id	Operator	Entry
T_6	crossover	$\langle \mathrm{ID}(T_1), \mathrm{ID}(T_4), \mathrm{ID}(R_1) \rangle$
T_7	crossover	$\langle \mathrm{ID}(T_4), \mathrm{ID}(T_5), \mathrm{ID}(R_2) \rangle$
T_8	crossover	$\langle \mathrm{ID}(T_3), \mathrm{ID}(T_5), \mathrm{ID}(R_3) \rangle$
T_9	crossover	$\langle \mathrm{ID}(T_1), \mathrm{ID}(T_5), \mathrm{ID}(R_4) \rangle$
T_{10}	crossover	$\langle \mathrm{ID}(T_3), \mathrm{ID}(T_4), \mathrm{ID}(R_5) \rangle$

(c)

Fig. 8.12 Illustration of the example described in Section 8.5.5. (a) The initial population P; (b) The random trees used by crossover; (c) The representation in memory of the new population P'

store the evaluations we create a table (which we call $\mathscr{V}$ from now on) containing, for each individual in $\mathscr{P}$, the values resulting from its evaluation on each fitness case (in other words, it contains the semantics of that individual). Hence, with a population of n individuals and a training set of k fitness cases, table $\mathscr{V}$ will be made of n rows and k columns. Then, for every generation, a new empty table $\mathscr{V}'$ is created. Whenever a new individual T must be generated by crossover between selected parents T_1 and T_2, T is represented by a triplet $T = \langle \mathrm{ID}(T_1), \mathrm{ID}(T_2), \mathrm{ID}(R) \rangle$, where R

is a random tree and, for any tree τ, $\mathrm{ID}(\tau)$ is a *reference* (or memory pointer)[7] to τ (using a C-like notation). This triplet is stored in an appropriate structure (which we call $\mathscr{M}$ from now on) that also contains the name of the operator used, as shown in Figure 8.12(c). The random tree R is created, stored in $\mathscr{P}$, and evaluated in each fitness case to reveal its semantics. The values of the semantics of T are also easily obtained, by calculating $(T_1 \cdot R) + ((1-R) \cdot T_2)$ for each fitness case, according to the definition of geometric semantic crossover, and stored in $\mathscr{V}'$. Analogously, whenever a new individual T must be obtained by applying mutation to an individual T_1, T is represented by a triplet $T = \langle \mathrm{ID}(T_1), \mathrm{ID}(R_1), \mathrm{ID}(R_2) \rangle$ (stored in $\mathscr{M}$), where R_1 and R_2 are two random trees (newly created, stored in $\mathscr{P}$ and evaluated for their semantics). The semantics of T is calculated as $T_1 + ms \cdot (R_1 - R_2)$ for each fitness case, according to the definition of geometric semantic mutation, and stored in $\mathscr{V}'$. At the end of each generation, table $\mathscr{V}'$ is copied into $\mathscr{V}$ and erased. All the rows of $\mathscr{P}$ and $\mathscr{M}$ referring to individuals that are not ancestors[8] of the new population can also be erased. Note that, while $\mathscr{M}$ grows at every generation, by keeping the semantics of the individuals separated we are able to use a table $\mathscr{V}$ whose size is independent of the number of generations. Summarizing, this algorithm is based on the idea that, when semantic operators are used, an individual can be fully described by its semantics (which makes the syntactic component much less important than in standard GP), a concept discussed in depth in [Moraglio et al., 2012]. Therefore, at every generation we update table $\mathscr{V}$ with the semantics of the new individuals, and save the information needed to build their syntactic structures without explicitly building them.

In terms of computational time, it is worth emphasizing that the process of updating table $\mathscr{V}$ is very efficient as it does not require the evaluation of the entire trees. Indeed, evaluating each individual requires (except for the initial generation) constant time, which is independent of the size of the individual itself. In terms of memory, tables $\mathscr{P}$ and $\mathscr{M}$ grow during the run. However, table $\mathscr{P}$ adds a maximum of $2 \times n$ rows per generation (if all new individuals are created by mutation) and table $\mathscr{M}$ (which contains only memory pointers) adds a maximum of n rows per generation. Even if we never erase the "ex-ancestors" from these tables (and never reuse random trees, which is also possible), we can manage them efficiently for several thousands of generations. Let us briefly consider the cost in terms of time and space of evolving a population of n individuals for g generations. At every generation, we need $O(n)$ space to store the new individuals. Thus, we need $O(ng)$ space in total. Since we need to do only $O(1)$ operations for any new individual (since the fitness can be computed using the fitness of the parents), the time complexity is also $O(ng)$. Thus, we have a linear space and time complexity with respect to

[7] Simple references to *lookup table* entries can be used in the implementation instead of real memory pointers (see [Vanneschi et al., 2013, Castelli et al., 2015]). This makes the implementation possible also in programming languages that do not allow direct manipulation of memory pointers, for instance Java or MatLab.

[8] The term "ancestors" here is abused slightly to designate not only the parents but also the random trees used to build an individual by crossover or mutation.

population size and number of generations. This computational complexity is very reasonable, and for sure competitive with several other ML systems.

The final step of the algorithm is performed after the end of the last generation. In order to reconstruct the individuals, we may need to "unwind" the compact representation and make the syntax of the individuals explicit. Therefore, despite performing the evolutionary search very efficiently, in the end we may not avoid dealing with the large trees that characterize the standard implementation of geometric semantic operators. However, most probably we will only be interested in the best individual found, so this unwinding (and recommended simplification) process may be required only once, and it is done offline after the run is finished. This greatly contrasts with the solution proposed by Moraglio et al. of building and simplifying *every* tree in the population at each generation online with the search process. If we are not interested in the form of the optimal solution, we can avoid the "unwinding phase" and we can evaluate an unseen input with a time complexity equal to $O(ng)$. In this case the individual is used as a "black box" which, in some cases, may be sufficient. Excluding the time needed to build and simplify the best individual, the proposed implementation allows us to evolve populations for thousands of generations with a considerable speed up with respect to standard GP.

Example 8.4. Let us consider the simple initial population P shown in table (a) of Figure 8.12 and the simple pool of random trees that are added to P as needed, shown in table (b). For simplicity, we will generate all the individuals in the new population (which we call P' from now on) using only crossover, which will require only this small amount of random trees. Besides the representation of the individuals in infix notation, these tables contain an identifier (Id) for each individual (T_1, ..., T_5 and R_1, ..., R_5). These identifiers will be used to represent the different individuals, and the individuals created for the new population will be represented by the identifiers T_6, ..., T_{10}. The individuals of the new population P' are simply represented by the set of entries exhibited in table (c) of Figure 8.12. This table contains, for each new individual, a *reference* to the ancestors that have been used to generate it and the name of the operator used to generate it (either "crossover" or "mutation"). For example, the individual T_6 is generated by the crossover of T_1 and T_4 and using the random tree R_1.

Let us assume that now we want to reconstruct the genotype of one of the individuals in P', for example T_{10}. The tables in Figure 8.12 contain all the information needed to do that. In particular, from table (c) we learn that T_{10} is obtained by crossover between T_3 and T_4, using random tree R_5. Thus, from the definition of geometric semantic crossover, we know that it will have the following structure: $(T_3 \cdot R_5) + ((1 - R_5) \cdot T_4)$. The remaining tables (a) and (b), which contain the syntactic structure of T_3, T_4, and R_5, provide us with the rest of the information we need to completely reconstruct the syntactic structure of T_{10}, which is:

$$((x_3 + x_4 - 2\,x_1) \cdot (2\,x_1)) + ((1 - (2\,x_1)) \cdot (x_1\,x_3))$$

and upon simplification becomes:

$$-x_1\,(4\,x_1 - 3\,x_3 - 2\,x_4 + 2\,x_1\,x_3).$$

8.5.6 Learning and Generalization in Real-Life Applications

The literature reports on various results obtained on several different domains using the implementation of GSGP presented above. In this section, a subset of those results is briefly discussed. The objective of this section is only to give the reader an idea of the quality of the results that can be obtained using GSGP, including a discussion of its generalization ability. We do not intend in any way to be exhaustive about the results that have been obtained. For a deeper review of the literature, the reader is referred to [Vanneschi et al., 2014].

All the applications presented in this section are real-life symbolic regression problems. Table 8.2 summarizes the main characteristics of each one of them. As

Table 8.2 The main characteristics of the test problems used in the experiments presented in this section

Dataset Name (ID)	# Features	# Instances	Objective
%F	241	359	Predicting the value of human oral bioavailability of a candidate new drug as a function of its molecular descriptors
%PPB	626	131	Predicting the value of the plasma protein binding level of a candidate new drug as a function of its molecular descriptors
LD50	626	234	Predicting the value of the toxicity of a candidate new drug as a function of its molecular descriptors
PEC	45	240	Predicting the value of energy consumption in one day as a function of a set of meteorologic data, and other kinds of data, concerning that day
PARK	18	42	Predicting the value of Parkinson's disease severity as a function of a set of the patient's data
CONC	8	1028	Predicting the value of concrete strength as a function of a set of features of concrete mixtures

the table shows, we are using six different problems. The first three of them (%F, %PPB and LD50) are problems in pharmacokinetics and they consist in the prediction of a pharmacokinetic parameter (bioavailability, plasma protein binding level and toxicity, respectively) as a function of some molecular descriptors of a potential new drug. The PEC dataset has the objective of predicting energy consumption in one day as a function of several different types of data relative to the previous days. The PARK dataset contains data about a set of patients with Parkinson's disease and the target value is a quantification of the severity of the disease, according to a standard measure. Finally, the CONC dataset has the objective of predicting

concrete strength as a function of a set of parameters that characterize a concrete mixture. For different reasons, all these problems are considered important in their respective domains, and they have been studied so far using several different computational intelligence methods.

The results that have been obtained concerning a comparison between GSGP and standard GP (ST-GP) on these problems are presented in Figure 8.13 (%F, %PPB and LD50 datasets) and Figure 8.14 (PEC, PARK and CONC datasets). The plots in these figures report, for each problem, the results obtained on the training set (leftmost plot) and on the test set (rightmost plot). A detailed discussion of the parameters used in both GP systems in these experiments is beyond the objective of this chapter. What we can generally observe from Figure 8.13 and Figure 8.14 is that GSGP is able to consistently outperform ST-GP both on the training set and on the test set for all the considered problems. Statistical tests also indicated that these differences are statistically significant.

The good results that GSGP has obtained on training data were expected: the geometric semantic operators induce a unimodal fitness landscape, which facilitates evolvability. On the other hand, on a first analysis, it has been a surprise to observe the excellent results that have been obtained on test data. These results even appeared a bit counterintuitive at first sight: we were expecting that the good evolvability on training data would entail an overfitting of those data. However, in order to give an interpretation to the generalization ability of GSGP, one characteristic of geometric semantic operators was realized that was not so obvious previously:

> *the geometric properties of geometric semantic operators hold independently of the data on which individuals are evaluated, and thus they hold also on test data!*

In other words, for instance, geometric semantic crossover produces an offspring that stands between the parents also in the semantic space induced by test data. As a direct implication, following exactly the same argument as Moraglio et al. [Moraglio et al., 2012], each offspring is, in the worst case, not worse than the worst of its parents on the test set. Analogously, as happens for training data, geometric semantic mutation produces an offspring that is a "weak" perturbation of its parent also in the semantic space induced by test data (and the maximum possible perturbation is, again, limited by the *ms* step). The immediate consequence for the behavior of GSGP on test data is that, while geometric semantic operators do not guarantee an improvement in test fitness each time they are applied, they at least guarantee that the possible worsening of the test fitness is bounded (by the test fitness of the worst parent for crossover, and by *ms* for mutation). In other words, *geometric semantic operators help control overfitting*. Of course, overfitting may still happen for GSGP (as happens, in a slight but visible way for instance in plots (d) and (f) of Figure 8.13, reporting the results on %PPB and LD50 respectively), but there are no big "jumps" in test fitness like the ones observed for ST-GP. It is worth remarking that, without the novel implementation presented in Section 8.5.5 that allowed us to use GSGP on these complex real-life problems, this interesting property would probably have remained unnoticed. We also remark that the implementation of GSM described in Section 8.5.3 contains one important de-

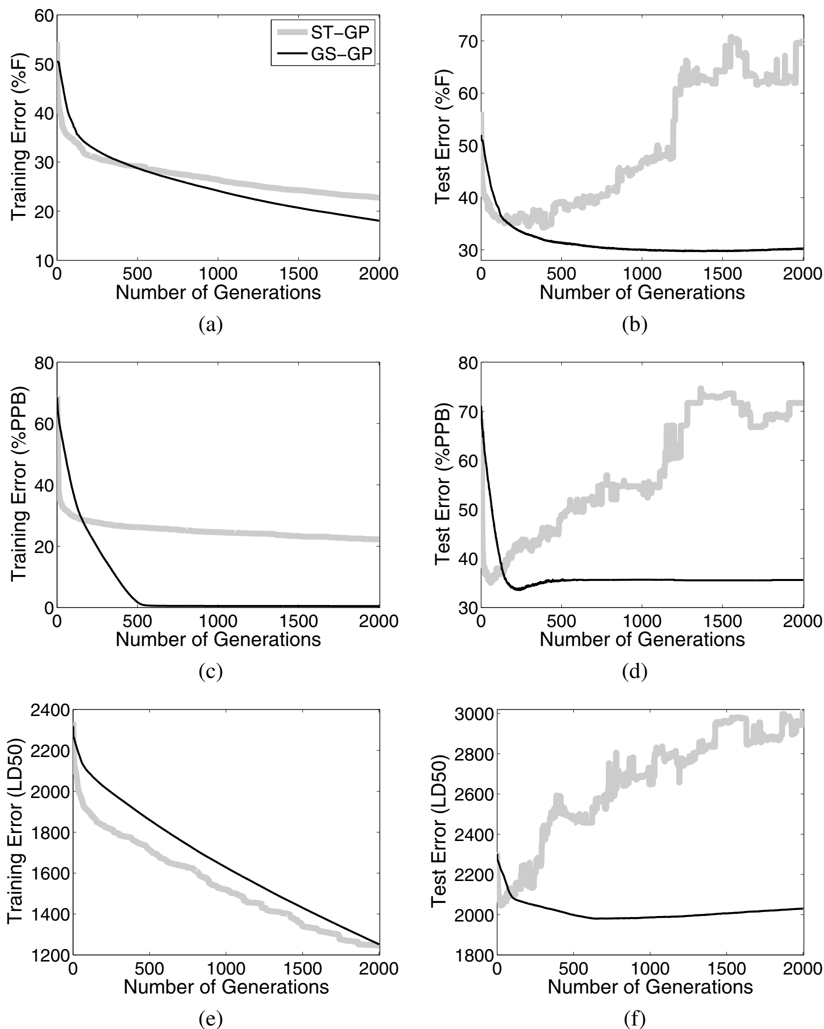

Fig. 8.13 Experimental comparison between standard GP (ST-GP) and Geometric Semantic GP (GS-GP). (a) %F problem, results on the training set; (b) %F problem, results on the test set; (c) %PPB problem, results on the training set; (d) %PPB problem, results on the test set; (e) LD50 problem, results on the training set; (f) LD50 problem, results on the test set

tail that was not in the original implementation of [Moraglio et al., 2012], which is the logistic function applied to the output of each of the random trees. Without this, the perturbation applied to the offspring of GSM would not be bound by the mutation step *ms*. Indeed, Gonçalves et al. [Gonçalves et al., 2015] have discovered that, without these bounds, GSGP overfits very quickly. The same work also proposes

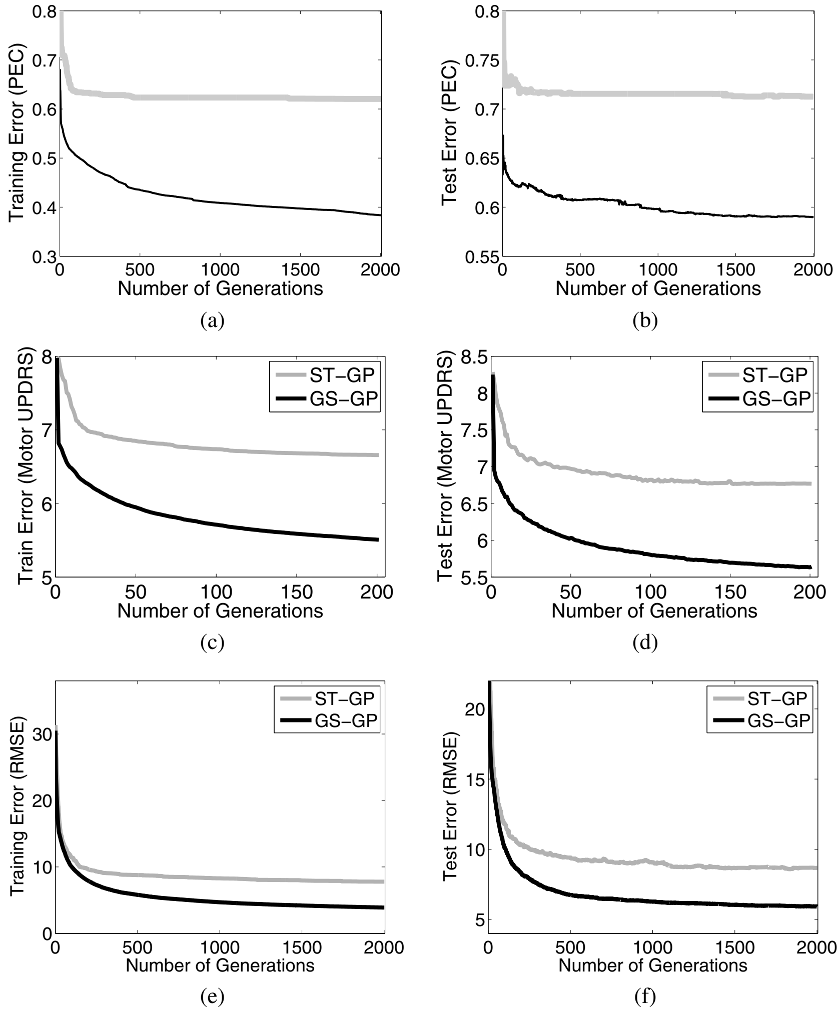

Fig. 8.14 Experimental comparison between standard GP (ST-GP) and Geometric Semantic GP (GS-GP). (a) PEC problem, results on the training set; (b) PEC problem, results on the test set; (c) PARK problem, results on the training set; (d) PARK problem, results on the test set; (e) CONC problem, results on the training set; (f) CONC problem, results on the test set

two variants of GSM that deterministically and optimally adapt the mutation step, obtaining competitive generalization in only one or two generations.

Table 8.3 also reports an experimental comparison between GSGP and a wide set of nonevolutionary machine learning state-of-the-art methods on the CONC dataset. Analogous results can be found in the literature also for all the other studied prob-

Table 8.3 An experimental comparison between GSGP (last line), ST-GP (second to last line) and other machine learning strategies on the CONC dataset. The leftmost column contains the name of the method, the middle one the results obtained on the training set at termination, and the rightmost column contains the results obtained on the test set. Root Mean Square Error (RMSE) results are reported

Method	Train	Test
Linear regression	10.567	10.007
Square Regression	17.245	15.913
Isotonic Regression	13.497	13.387
Radial Basis Function Network	16.778	16.094
SVM Polynomial Kernel (1st degree)	10.853	10.260
SVM Polynomial Kernel (2nd degree)	7.830	7.614
SVM Polynomial Kernel (3rd degree)	6.323	6.796
SVM Polynomial Kernel (4th degree)	5.567	6.664
SVM Polynomial Kernel (5th degree)	4.938	6.792
Artificial Neural Networks	7.396	7.512
Standard GP	7.792	8.67
Geometric Semantic GP	3.897	5.926

lems, and they show that GSGP is able to outperform all the other techniques both on training and test data.

8.6 Multiclass GP

Until now, classification has not been discussed in the context of GP. Looking back at Section 8.3.1, pertaining to symbolic regression with GP, it is easy to see why many binary classification problems are solved as if they, too, were regression problems [Espejo et al., 2010].

The definition of a classification problem is very similar to that of a regression problem. Given a set of vectors $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n\}$, where for all $i = 1, 2, ..., n, \mathbf{x}_i \in \mathbb{R}^p$, and a categorical vector $\mathbf{y} = [y_1, y_2, ..., y_n]$, where for all $i = 1, 2, ..., n, y_i \in M$, where M is a set of class labels, a classification problem can be generally defined as the problem of finding a mapping $g : \mathbb{R}^p \to M$ such that $\forall i = 1, 2, ..., n : g(\mathbf{x}_i) = y_i$.

All it takes to transform a binary classification problem into a regression problem is to represent the two class labels as numeric expected outputs, e.g., 0 and 1, so that GP can run as usual, using the same function and terminal sets, and the same fitness function that would be used for symbolic regression. With this setting, GP will find a function that outputs values close to 0 for one class, and close to 1 for the other class. In order to obtain class predictions from such a model, a cutoff is applied to the predicted numeric outputs, e.g., 0.5. Predictions below the cutoff are labeled as class A, while predictions above the cutoff are labeled as class B. Figure 8.15 illustrates this setting, where one data point belonging to class B (yellow crosses)

would be misclassified as class A (blue circles) because its value is lower than the cutoff.

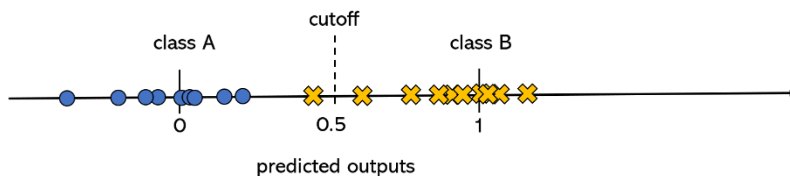

Fig. 8.15 Representation of regression-like binary classification, by transforming class labels A and B into numeric expected outputs 0 and 1, and applying a cutoff of 0.5 to the numeric predicted outputs to obtain label predictions. Class A data points are represented as blue circles, while class B data points are represented as yellow crosses

Another, more relaxed option is to allow complete freedom regarding the predicted output values and use a fitness function that simply rewards the minimization of the overlap between whichever ranges are produced for each class. For future reference, we designate this method as *free_outputs*. We remark that the function evolved to produce the output values is a combination of the p original features of the problem, and as such can be called a hyperfeature. In Figure 8.15 there is no overlap between the ranges of the two classes.

In order to classify unseen data using the *free_outputs* approach, we need more than just the evolved function, or hyperfeature. Just as earlier we needed to know that 0.5 was the cutoff to apply to the predicted outputs, now we also need to know how to cast each predicted output into a class. One option is to, once again, provide a cutoff, this time calculated to maximize the accuracy on the training data. Another simple option is to provide the mean points of each class, so that each unseen data point is assigned to the class with the nearest mean.

What about multiclass classification problems? It is certainly tempting to use the same rationale of transforming class labels into numeric expected outputs and obtaining class predictions by applying multiple cutoffs to the predicted outputs. However, it is too difficult to find a single function that can handle three or more unrelated classes and produce distinct outputs for each one of them. Minimizing the overlap between class ranges is equally difficult. Naturally, it is possible to solve classification problems with GP in a non-regression-like fashion, for example, by using conditional operators to evolve models similar to decision trees, among other options described in [Espejo et al., 2010].

One recent family of methods, here collectively designated as Multiclass GP, takes the *free_outputs* classification method one step further. Instead of evolving a single hyperfeature, it collectively evolves a number of hyperfeatures that effectively produce a mapping between the original p-dimensional feature space and a new q-dimensional hyperfeature space, where q is independent of p and of the number of classes m. Figure 8.16 illustrates a two-dimensional space containing points of three different classes. In this new space, Multiclass GP applies a nearest centroid

classifier based on the Mahalanobis distance, using an accuracy-based measure as fitness.

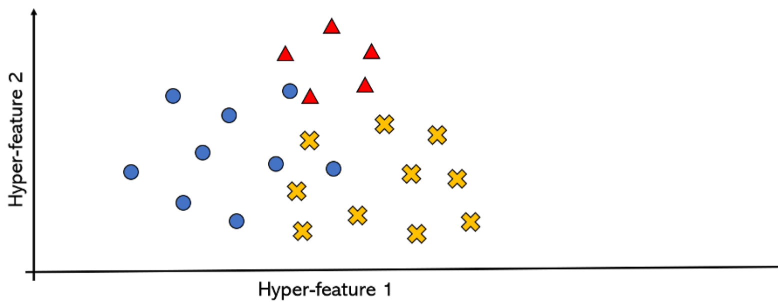

Fig. 8.16 Representation of a two-dimensional hyperfeature space containing data points belonging to three different classes, represented as blue circles, yellow crosses and red triangles

In the remainder of this chapter we will present detailed descriptions of four variants of Multiclass GP, named M2GP, M3GP, eM3GP and M4GP, including results obtained in different classification datasets, whose characteristics are summarized in Table 8.4. We will finish with a discussion regarding the use of Multiclass GP as a feature construction method.

Table 8.4 Datasets used for comparing different Multiclass GP variants. The 'heart' (HRT), 'segment' (SEG), 'vowel' (VOW), 'yeast' (YST) and 'movement-libras' (M-L) datasets can be found at the KEEL repository [Alcala-Fdez et al., 2011], whereas the 'waveform' (WAV) dataset is available at [Bache and Lichman, 2013]. 'IM-3' and 'IM-10' are the satellite imagery datasets used in [Vasconcelos et al., 2015, Batista and Silva, 2020, Batista et al., 2021]

Dataset	**HRT**	**IM-3**	**WAV**	**SEG**	**IM-10**	**YST**	**VOW**	**M-L**
Classes	2	3	3	7	10	10	11	15
Features	13	6	40	19	6	8	13	90
Samples	270	322	5000	2310	6798	1484	990	360

8.6.1 M2GP – Multidimensional Multiclass GP

The basic idea of M2GP, originally introduced by [Ingalalli et al., 2014] as Multidimensional Multiclass GP, is to find a transformation such that the transformed data can be grouped in clusters, one cluster per class. As stated earlier, the number of dimensions of the transformed data is independent of the number of dimensions of the original data and of the number of classes. Therefore, it may happen that a high-dimensional dataset containing many classes is easily classified by a low-

dimensional clustering, while a low-dimensional dataset with few classes is better classified by a high-dimensional clustering. The components of M2GP that are different from those of a standard GP implementation (Section 8.1) are described below.

Representation. M2GP uses a representation for the solutions that allows them to perform the mapping $k : \mathbb{R}^p \rightarrow \mathbb{R}^q$. The representation is basically the same used for regular tree-based GP, except that the root node of the tree is a dummy node that exists only to define the number of dimensions q of the new space. Each branch stemming directly from the root encodes one hyperfeature that performs the mapping in one of the q dimensions.

Initialization. M2GP trees are initialized using ramped half-and-half skewed to 25% Grow and 75% Full based on the suggestion that a higher proportion of full trees facilitates the initial evolution [Ingalalli et al., 2014]. Below, we explain how to establish the number of branches, i.e., dimensions of the hyperfeature space.

Fitness Evaluation. The truly specialized element of M2GP is the fitness function. Each individual is evaluated in the following way. All the p-dimensional data points of the training set are mapped into the new q-dimensional space (each branch of the tree encodes one of the q dimensions). On this new space, for each of the m classes in the dataset, the covariance matrix and the cluster centroid are calculated from the samples belonging to that class. The Mahalanobis distance between each point and each of the m centroids is calculated. Each point is assigned the class whose centroid is closer. Fitness is the overall accuracy of this classification (the percentage of samples correctly classified).

The Mahalanobis distance between a data point $\mathbf{x}_i$ and the centroid $\mathbf{c}_j$ of the cluster formed by the points of class j is given by

$$D_M = \sqrt{(\mathbf{x}_i - \mathbf{c}_j)\Sigma^{-1}(\mathbf{x}_i - \mathbf{c}_j)},$$

where Σ is the covariance matrix of the points belonging to class j. In M2GP, whenever Σ is not invertible, the Euclidean distance is used instead.

The preference for Mahalanobis instead of Euclidean distance is not a small detail. Initial studies have consistently shown that the distance metric indeed plays a substantial role in the performance of M2GP, especially in higher-dimensional solution spaces [Ingalalli et al., 2014]. Unlike the Euclidean distance, the Mahalanobis distance not only is able to capture the physical distance between the sample and the class clustered data sets, but also considers the statistical correlation between them, thereby reasserting the work of [Shiming Xiang and Zhang, 2008].

Genetic Operators. The genetic operators used by M2GP are the regular subtree crossover and mutation, used in [Ingalalli et al., 2014] with probabilities 0.9 and 0.1, respectively, except that the root of the tree (the dummy node) is never chosen as the crossing or mutation node.

At the end of a run, the solution returned by M2GP is composed not only of the

tree of the best individual, but also of the covariance matrices and cluster centroids of each class. In order to classify unseen data, M2GP uses the tree to map the new samples into the new space, and then uses the covariance matrices and the cluster centroids in order to determine the minimum Mahalanobis distance between each sample and each centroid. (Note that the covariance matrices and cluster centroids are not recalculated when classifying new data.)

Choosing the number of dimensions. M2GP is incapable of adding or removing dimensions during the evolution, so the number of dimensions q has to be fixed at the beginning of each run. It is not obvious how to choose this important parameter, as it will naturally depend on the characteristics of each problem, as well as on the other GP settings such as the function and terminal sets. However, [Ingalalli et al., 2014] noticed that the best fitness found on the initial random generation is highly correlated with the best fitness found on the final generation. Therefore, before initiating a run, M2GP chooses q by executing the following procedure:

Initialization:	$q = 1$; create a random initial population where all the individuals have only one branch, and record the fitness of the best individual;
In each iteration:	create a new initial population where all the individuals have $q+1$ branches; if the fitness of the new best individual is better than the previously recorded one, then $q = q+1$;
Stop condition:	q was not incremented.

Table 8.5 shows a subset of the results[9] reported in [Ingalalli et al., 2014], comparing the median test accuracy of different methods on the datasets listed in Table 8.4. In most problems, M2GP is surpassed by at least one other method. However, unlike all other methods, M2GP is never the worst method on any of the datasets. Therefore, M2GP appears to be a safe and competitive classifier.

8.6.2 M3GP – M2GP with Multidimensional Populations

As described above, the original M2GP uses a greedy approach to determine how many dimensions the evolved solutions should have. It may happen that, by fixing the number of dimensions at the beginning of the run, the algorithm is prevented from finding better solutions during the search, ones that may use a different number of dimensions. Therefore, a new variant of Multiclass GP, originally presented in [Muñoz et al., 2015], uses additional genetic operators that add, remove and swap dimensions. The result is a population that includes individuals of different dimensions, where natural selection discards the worst ones and naturally leads the search

[9] The results reported here are different from [Ingalalli et al., 2014] in two cases:
1) In [Ingalalli et al., 2014] the result of M2GP for WAV was incorrectly reported as 94.8;
2) For reasons stated in [Ingalalli et al., 2014], the results reported for HRT referred to standard GP and not M2GP.

Table 8.5 Comparison between M2GP and state-of-the-art methods. Median test accuracy obtained in 30 runs on the datasets listed in Table 8.4. For each problem, the best results are in bold (more than one means there is no statistically significant difference between their medians). For each problem, a plus sign after the value means the method was significantly better than M2GP, while a minus sign means the method was significantly worse than M2GP. The statistical test used was Kruskal-Wallis with Bonferroni correction at the 0.01 significance level

	HRT	IM-3	WAV	SEG	IM-10	YST	VOW	M-L
SVM	55.6−	**93.8**	**86.3**+	55.8−	90.4	41.1−	81.8−	14.4−
J48	79.6	**93.8**	74.8−	96.1+	94.7+	55.2	75.9−	63.4
RF	80.2	**94.8**	81.5−	**97.3**+	**96.9**+	**57.5**+	**89.4**+	71.8+
RS	**81.5**	**92.8**	82.2−	96.0	93.9+	56.6	82.8	65.7
MLP	80.2	**95.9**	83.3−	96.3+	90.2	**58.0**+	82.5−	**75.9**+
MCC	**84.0**+	**95.4**	**86.8**+	92.4−	81.8−	**58.0**+	57.6−	60.6
M2GP	80.2	**93.8**	84.9	95.6	90.2	53.8	85.9	63.0

towards the ideal number of dimensions. This variant is called M3GP, which stands for M2GP with Multidimensional Populations. In M3GP, the representation of the individuals is the same as in M2GP, as well as the fitness evaluation. The next paragraphs explain how M3GP initializes the population, and describe the new genetic operators it uses, including a pruning operator, and finally explain why these particular aspects make elitism an important factor in M3GP.

Initialization. M3GP starts the evolution with a random population where all the individuals have only one dimension. This ensures that the evolutionary search begins looking for simple, unidimensional solutions, before moving towards higher-dimensional and potentially more complex solutions. To avoid individuals that are too small to be useful, and since M2GP already biased the initial population toward full trees, M3GP uses only the Full initialization method to create all its initial individuals.

Mutation. During the breeding phase, whenever mutation is the chosen genetic operator, M3GP performs one of three actions, with equal probability: 1) standard subtree mutation, taking care not to touch the root node, as in M2GP; 2) adding a randomly created new tree as a new branch of the root node, effectively adding one dimension to the parent tree; 3) randomly removing a complete branch of the root node, effectively removing one dimension from the parent tree. As mentioned above, the initial population only contains unidimensional individuals. From there, the algorithm has to be able to explore several different dimensions. As mutation is the only way of adding or removing dimensions, in M3GP it assumes a high importance and therefore its probability of occurrence is 50% (i.e., 50/3% for each of the three mutation types).

Crossover. Whenever crossover is chosen, M3GP performs one of two actions, with equal probability: 1) standard subtree crossover, avoiding the root node as in M2GP; 2) swapping of dimensions, where a random complete branch of the root node is chosen in each parent, and swapped with each other, effectively swapping dimen-

sions between the parents. The second event is just a particular case of the first, where the crossing nodes are guaranteed to be directly connected to the root node. The probability of occurrence of crossover is 50%, therefore 25% for each of the two crossover types.

Pruning. Mutation, as described above, makes it easy for M3GP to add dimensions to the solutions. However, many times some of the dimensions are useless or even degrade the fitness of the individual, so they would be better removed. Mutation can also remove dimensions but, as described above, it does so randomly and blind to fitness. Instead of making the genetic operators more 'intelligent', M3GP keeps them simple and completely stochastic, while relying on a pruning operator to trim unwanted dimensions. The pruning procedure removes the first dimension and reevaluates the tree. If the fitness improves, the pruned tree replaces the original and goes through pruning of the next dimension. Otherwise, the pruned tree is discarded and the original tree goes through pruning of the next dimension. The procedure stops after pruning the last dimension. Pruning is applied only to the best individual in each generation. Applying it to all the individuals in the population could pose two problems: 1) a significantly higher computational demand, where a considerable amount of effort would be spent on individuals that would still be unfit after pruning; 2) the possibility of causing premature convergence due to excessive removal of genetic material (the same way that code editing has been shown to cause it [Haynes, 1998]. Preliminary experiments in [Muñoz et al., 2015] have revealed that pruning the best individual of each generation shifts the distribution of the number of dimensions to lower values (or prevents it from shifting to higher values so easily) during the evolution, without harming fitness.

Elitism. As mentioned above, in order to explore solutions of different dimensions M3GP relies on mutation to add and remove dimensions from the individuals, with a fairly high probability. It also has to rely on selection to keep the best dimensions in the population and discard the worst ones. The way to do this is by ensuring some elitism in the survival of the individuals from one generation to the next. Unlike M2GP, where elitism was mostly inconsequential to the outcome of the evolution, M3GP cannot afford to lose the best individual of any generation, and therefore always copies it to the next generation. We recall that this individual is already optimized in the sense that it went through pruning.

Table 8.6 shows the results reported in [Silva et al., 2016] comparing the performance of M2GP and M3GP (and eM3GP, to be discussed later) on the datasets listed in Table 8.4, in terms of training and test accuracy, and number of nodes and dimensions of the solutions returned. In training, M3GP surpasses M2GP in all the problems except M-L, where the two methods perform equally well. In test, M3GP is better than or equal to M2GP on all problems except M-L. It is interesting to note that it is on the higher-dimensional problems (except M-L) that M3GP achieves better results than M2GP (the problems are roughly ordered by dimensionality of the data). The M-L exception may have a simple explanation, which is how eas-

ily M3GP reaches maximal accuracy on the training set. Both M2GP and M3GP achieve 100% training accuracy, but M3GP does it in only a few generations, producing very small and accurate solutions that barely generalize to unseen data. On the other hand, M2GP does not converge immediately, so in its effort to learn the characteristics of the data it also evolves some generalization ability. Regarding the structure and size of the returned models, M3GP tends to use more dimensions and more nodes than M2GP.

Table 8.6 Comparison between M2GP, M3GP and eM3GP. Median values of training and test accuracy, and number of nodes and dimensions (with minimum and maximum), obtained in 30 runs on the datasets listed in Table 8.4. For each problem, the best fitness results are in bold (more than one means there is no statistically significant difference between their medians). The statistical test used was Friedman with Bonferroni-Holm correction at the 0.05 significance level

	HRT	**IM-3**	**WAV**	**SEG**	**IM-10**	**YST**	**VOW**	**M-L**
Training fitness								
M2GP	89.4	98.2	87.4	96.8	91.4	62.6	95.9	**100**
M3GP	**94.7**	**99.6**	**90.7**	**98.1**	**93.0**	**68.5**	**100**	**100**
eMeGP	86.7	98.2	81.8	96.1	92.0	61.0	87.8	**100**
Test fitness								
M2GP	**80.2**	**93.8**	**84.9**	**95.6**	90.2	53.8	85.9	**63.0**
M3GP	**79.0**	**95.4**	**84.3**	**95.6**	**91.0**	**56.2**	**93.8**	57.1
eM3GP	**80.8**	**93.2**	81.2	94.7	**90.3**	**56.1**	78.6	**65.1**
Number of nodes								
M2GP	37	24	126	43	117	146	49	33
M3GP	110	66	71	111	239	274	53	13
eM3GP	4	8	3	8	58	14	10	4
Number of dimensions								
M2GP	2.5 *(1-8)*	2 *(1-4)*	5 *(2-10)*	4 *(3-8)*	7 *(4-10)*	5.5 *(1-13)*	9 *(4-18)*	10 *(7-12)*
M3GP	12 *(1-17)*	5 *(2-8)*	31 *(29-37)*	11 *(5-21)*	12 *(11-16)*	13 *(11-18)*	20 *(16-20)*	12 *(10-13)*
eM3GP	1 *(1-4)*	1 *(1-5)*	1 *(1-10)*	6 *(2-10)*	7 *(3-12)*	10 *(1-16)*	4 *(1-14)*	2 *(1-11)*

Table 8.7 shows the results reported in [Silva et al., 2016] comparing the training and test accuracy of different methods on the datasets listed in Table 8.4. In test accuracy, RF was the best method on five datasets, followed by M3GP and MLP, the best methods on four datasets each.

8.6.3 eM3GP – Ensemble M3GP

The eM3GP variant, proposed in [Silva et al., 2016], stands for Ensemble M3GP and was developed in an attempt to avoid two problems observed in M3GP: 1) the occasional degradation of class accuracy on some classes that were previously well classified, caused by the discovery of new best individuals that improve overall accuracy based on other classes; 2) the proliferation of dimensions that happens on some problems, resulting in much larger individuals, but not much fitter, than the ones obtained with M2GP.

Table 8.7 Comparison between M2GP, M3GP, eM3GP and state-of-the-art methods. Median training and test accuracy obtained in 30 runs on the datasets listed in Table 8.4. For each problem, the best results are in bold (more than one means there is no statistically significant difference between their medians). The statistical test used was Friedman with Bonferroni-Holm correction at the 0.05 significance level

	HRT	IM-3	WAV	SEG	IM-10	YST	VOW	M-L
Training fitness								
RF	**98.4**	**100**	**99.5**	**99.9**	**99.8**	**98.3**	99.9	99.2
RS	88.9	97.1	92.0	98.4	96.3	71.1	97.8	92.3
MLP	**98.4**	98.7	98.5	97.6	91.0	64.6	91.9	91.3
M2GP	89.4	98.2	87.4	96.8	91.4	62.6	95.9	**100**
M3GP	94.7	99.6	90.7	98.1	93.0	68.5	**100**	**100**
eM3GP	86.7	98.2	81.8	96.1	92.0	61.0	87.8	**100**
Test fitness								
RF	**80.2**	**94.8**	81.5	**97.3**	**96.9**	**57.5**	89.4	71.8
RS	**81.5**	92.8	82.2	96.0	93.9	56.6	82.8	65.7
MLP	**80.2**	**95.9**	83.3	96.3	90.2	**58.0**	82.5	**75.9**
M2GP	**80.2**	93.8	**84.9**	95.6	90.2	53.8	85.9	63.0
M3GP	**79.0**	**95.4**	**84.3**	95.6	91.0	56.2	**93.8**	57.1
eM3GP	**81.4**	93.2	81.2	94.7	90.3	56.1	78.6	65.1

Although similar to M3GP is many aspects, eM3GP assumes that a set of transformations, each proving to be good for discriminating a single class, can do a better job than a single transformation used for discriminating all the classes. Therefore, the main differences introduced by eM3GP are the following.

Storing specialized transformations. During an eM3GP run, a catalog of "specialized" transformations is kept, one for each class. When eM3GP begins its search and creates its very first individual, m copies of this transformation are stored as the set of best transformations for each of the m classes of the problem. The numbers of true positives (TP) and false positives (FP) obtained by this transformation on each class are also stored. With every new individual created, the TP and FP numbers are calculated and compared with the stored ones. For each class, if the new individual improves either of these numbers (increases TP or decreases FP) without degrading any of them, then the individual replaces the stored one as the best transformation for this class, while for the other classes the stored transformation remains unaltered. Note that these are not exactly "specialized" transformations, since they are not evolved specifically for binary classification. Note also that, at any given moment, among the m elements of this catalog there may be repeated individuals.

Representation. The individuals of eM3GP are created, recombined and mutated using the same representation and genetic operators as the individuals of M3GP. However, when it's time to use an individual to make a prediction (which is required for measuring its fitness) it acquires an ensemble-like representation. The individual is first replicated in m copies, one per class, and then combined with the catalog of m specialized transformations in an iterative nondeterministic process, with the goal of deciding which transformation (copy of individual or stored spe-

cialized transformation) is adopted for each class. First, fitness is measured for the ensemble of m copies of the same transformation. Then, in a random order, the copy of each class is replaced by the corresponding specialized transformation of the catalog, and fitness is measured again. If the replacement improves fitness, it becomes permanent, otherwise it is undone. In the end, among the m components of the final ensemble, there may or may not be a copy of the original individual. However, using only the catalog instead of a combination between individual and catalog proved to be detrimental to fitness [Silva et al., 2016].

Fitness Evaluation. The only difference between the fitness function of eM3GP and that of the previous variants is that the distances to the cluster centroids of each class are measured on the hyperfeature space obtained by the transformation adopted for that class.

Tables 8.6 and 8.7 include the results of eM3GP, together with the results of M2GP and M3GP, comparing them with each other and with other state-of-the-art methods on the datasets listed in Table 8.4. These results were reported in [Silva et al., 2016] and show that, although in most problems eM3GP was not able to surpass the accuracy of M3GP, in many cases a similar accuracy was obtained with much smaller individuals.

8.6.4 M4GP

Unlike the previous variants, which use a tree-based representation for the solutions, M4GP [La Cava et al., 2018, La Cava et al., 2019] uses a stack-based representation. By default, this representation supports multiple outputs, eliminating the need for the dummy root nodes required by the previous variants. M4GP initializes the population with individuals of various sizes and dimensionality, and applies genetic operators that are the stack-based equivalents to those of M3GP. Besides the representation and initialization, M4GP introduces additional differences to the selection of individuals for breeding and for survival. In [La Cava et al., 2019], M4GP with regular tournament selection [Poli et al., 2008a] (M4GP-tn) is compared with a variant that uses lexicase selection [Helmuth et al., 2015] (M4GP-lx) and another one that uses age-fitness Pareto survival [Schmidt and Lipson, 2011] (M4GP-ps). Table 8.8 shows the results[10] of comparing the test accuracy of these three variants of M4GP with the previous variants and the state-of-the-art methods on the datasets listed in Table 8.4. The superiority of M4GP is clear, in particular M4GP-ps.

[10] The results reported here are different from [La Cava et al., 2019] on two problems:
1) In the M-L problem (called Movl in [La Cava et al., 2019]) the results of RF, MLP, M2GP and eM3GP were incorrectly reported as 89.4, 82.5, 85.9 and 78.6, respectively;
2) For the YST (Yeast) problem, the result of M4GP-lx was incorrectly indicated as being statistically better than all the others.

Table 8.8 Comparison between M4GP and state-of-the-art methods. Median test accuracy obtained in 30 runs on the datasets listed in Table 8.4. For each problem, the best results are in bold (more than one means there is no statistically significant difference between their medians). The statistical test used was Wilcoxon rank-sum with Holm correction at the 0.01 significance level

	HRT	IM-3	WAV	SEG	IM-10	YST	VOW	M-L
RF	80.2	94.8	81.5	**97.3**	**96.9**	**57.5**	89.4	71.8
RS	81.5	92.8	82.2	96.0	93.9	**56.6**	82.8	65.7
MLP	80.2	95.9	83.3	96.3	90.2	**58.0**	82.5	75.9
SVM	55.6	93.8	86.3	55.8	90.4	41.1	81.8	14.4
M2GP	80.2	93.8	84.9	95.6	90.2	53.8	85.9	63.0
M3GP	79.0	95.4	84.3	95.6	91.0	**56.2**	93.8	57.1
eM3GP	80.9	93.3	81.2	94.7	90.3	56.2	78.6	65.2
M4GP-lx	85.2	**97.9**	85.3	**96.6**	90.7	**58.9**	95.6	73.1
M4GP-ps	**90.1**	**97.9**	**87.1**	96.1	89.8	**58.9**	**97.5**	**80.1**
M4GP-tn	87.7	**97.9**	86.0	95.1	89.6	**56.8**	96.0	**76.9**

8.6.5 Multiclass GP and Feature Construction

As stated above, a GP system that evolves combinations of the original features of a problem is actually building hyperfeatures, and therefore doing feature construction[11]. Of course the same can be said about other learners, but the fact that GP does not impose restrictions on the form of the allowed combinations makes it particularly adequate for knowledge discovery through feature construction, something that has been recognized since the early days of GP (e.g., [Bensusan and Kuscu, 1996]). The Multiclass GP variants described above further increase the potential of GP for feature construction. Although a nearest centroid classifier has been used in all variants, it can be easily replaced by any other classifier, naturally transforming the system into a powerful wrapper-like[12] or embedded method. Some recent publications, and references therein, highlight the potential of such systems for both classification and regression tasks [Muñoz et al., 2019, Muñoz et al., 2020, Cava and Moore, 2020, Batista and Silva, 2020, Batista et al., 2021].

Besides feature construction, M3GP has also been the inspiration for ensemble construction, in a system where regular trees are evolved by the standard genetic operators and, at the same time, forests are evolved by adding, deleting and swapping trees, using the same specialized operators as M3GP [Rodrigues et al., 2020].

[11] Also called feature learning, feature induction, feature extraction and feature discovery, among others, all part of the general concept of feature engineering (see Chapter 5).

[12] Normally, wrapper methods only do feature selection, but here both selection and construction can occur as a result of the same process.