43. Genetic Programming

James McDermott, Una-May O'Reilly

Genetic programming (GP) is the subset of evolutionary computation in which the aim is to create executable programs. It is an exciting field with many applications, some immediate and practical, others long-term and visionary. In this chapter, we provide a brief history of the ideas of genetic programming. We give a taxonomy of approaches and place genetic programming in a broader taxonomy of artificial intelligence. We outline some current research topics and point to successful use cases. We conclude with some practical GP-related resources including software packages and venues for GP publications.

43.1 Evolutionary Search for Executable Programs

There have been many attempts to artificially emulate human intelligence, from symbolic artificial intelligence (AI) [43.1] to connectionism [43.2, 3], to subcognitive approaches like behavioral AI [43.4] and statistical machine learning (ML) [43.5], and domain-specific achievements like web search [43.6] and self-driving cars [43.7]. Darwinian evolution [43.8] has a type of distributed intelligence distinct from all of these. It has created lifeforms and ecosystems of amazing diversity, complexity, beauty, facility, and efficiency. It has even created forms of intelligence very different from itself, including our own.

The principles of evolution – fitness biased selection and inheritance with variation – serve as inspiration for the field of evolutionary computation (EC) [43.9], an adaptive learning and search approach which is general-purpose, applicable even with black-box performance feedback, and highly parallel. EC is a trial-and-error method: individual solutions are evaluated for fitness, good ones are selected as parents, and new ones are created by inheritance with variation (Fig. 43.1).

GP is the subset of EC in which the aim is to create executable programs. The search space is a set of programs, such as the space of all possible Lisp programs within a subset of built-in functions and functions composed by a programmer or the space of numerical C functions. The program representation is an encoding of such a search space, for example an abstract syntax tree or a list of instructions. A program's fitness is evaluated by executing it to see what it does. New programs are created by inheritance and variation of material from

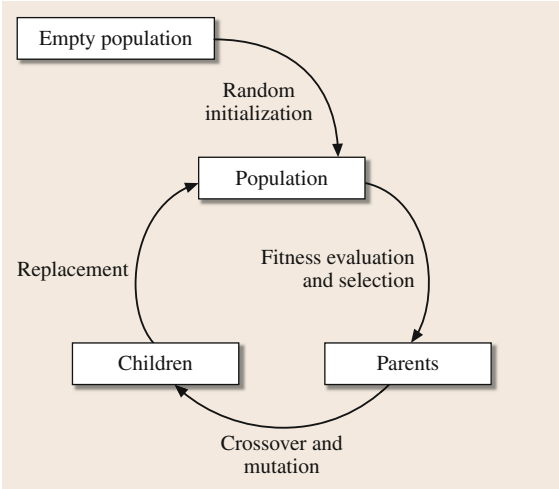

Fig. 43.1 The fundamental loop of EC

parent programs, with constraints to ensure syntactic correctness.

We define a program as a data structure capable of being executed directly by a computer, or of being compiled to a directly executable form by a compiler, or of interpretation, leading to execution of low-level code, by an interpreter. A key feature of some programming languages, such as Lisp, is *homoiconicity*: program code can be viewed as data. This is essential in GP, since when the algorithm operates on existing programs to make new ones, it is regarding them as data; but when they are being executed in order to determine what they do, they are being regarded as the program code. This double meaning echoes that of DNA (deoxyribonucleic acid), which is both data and code in the same sense.

GP exists in many different forms which differ (among other ways) in their executable representation. As in programming *by hand*, GP usually considers and composes programs of varying length. Programs are also generally hierarchical in some sense, with nesting of statements or control. These representation properties (variable length and hierarchical structure) raise a very different set of technical challenges for GP compared to typical EC.

GP is very promising, because programs are so general. A program can define and operate on any data structure, including numbers, strings, lists, dictionaries, sets, permutations, trees, and graphs [43.10–12]. Via Turing completeness, a program can emulate any model of computation, including Turing machines, cellular automata, neural networks, grammars, and finite-state machines [43.13–18].

A program can be a data regression model [43.19] or a probability distribution. It can express the growth process of a plant [43.20], the gait of a horse [43.21], or the attack strategy of a group of lions [43.22]; it can model behavior in the Prisoner's Dilemma [43.23] or play chess [43.24], Pacman [43.25], or a car-racing game [43.26]. A program can generate designs for physical objects, like a space-going antenna [43.27], or plans for the organization of objects, like the layout of a manufacturing facility [43.28]. A program can implement a rule-based expert system for medicine [43.29], a scheduling strategy for a factory [43.30], or an exam timetable for a university [43.31]. A program can recognize speech [43.32], filter a digital signal [43.33], or process the raw output of a brain-computer interface [43.34]. It can generate a piece of abstract art [43.35], a 3-D (three-dimensional) architectural model [43.36], or a piece of piano music [43.37].

A program can interface with natural or man-made sensors and actuators in the real world, so it can both act and react [43.38]. It can interact with a user or with remote sites over the network [43.39]. It can also introspect and copy or modify itself [43.40]. A program can be nondeterministic [43.41]. If true AI is possible, then a program can be intelligent [43.42].

43.2 History

GP has a surprisingly long history, dating back to very shortly after *von Neumann*'s 1945 description of the stored-program architecture [43.43] and the 1946 creation of ENIAC [43.44], sometimes regarded as the first general-purpose computer. In 1948, *Turing* stated the aim of machine intelligence and recognized that evolution might have something to teach us in this regard [43.45]:

> *Further research into intelligence of machinery will probably be very greatly concerned with searches.* [. . .] *There is the genetical or evolutionary search by which a combination of genes is looked for, the criterion being survival value. The remarkable success of this search confirms to some extent the idea that intellectual activity consists mainly of various kinds of search.*

However, Turing also went a step further. In 1950, he more explicitly stated the aim of automatic programming (AP) and a mapping between biological evolution and program search [43.46]:

> *We have* [...] *divided our problem [automatic programming] into two parts. The child-program [Turing machine] and the education process. These two remain very closely connected. We cannot expect to find a good child-machine at the first attempt. One must experiment with teaching one such machine and see how well it learns. One can then try another and see if it is better or worse. There is an obvious connection between this process and evolution, by the identifications:*

- *Structure of the child machine = Hereditary material*
- *Changes = Mutations*
- *Natural selection = Judgment of the experimenter.*

This is an unmistakeable, if abstract, description of GP (though a computational fitness function is not envisaged).

Several other authors expanded on the aims and vision of AP and machine intelligence. In 1959 *Samuel* wrote that the aim was to be able to *Tell the computer what to do, not how to do it* [43.47]. An important early attempt at implementation of AP was the 1958 *learning machine* of *Friedberg* [43.48].

In 1963, *McCarthy* summarized [43.1] several representations with which machine intelligence might be attempted: neural networks, Turing machines, and *calculator programs*. With the latter, McCarthy was referring to Friedberg's work. McCarthy was prescient in identifying important issues such as representations, operator behavior, density of good programs in the search space, sufficiency of the search space, appropriate fitness evaluation, and self-organized modularity. Many of these remain open issues in GP [43.49].

Fogel et al.'s 1960s *evolutionary programming* may be the first successful implementation of GP [43.50]. It used a finite-state machine representation for programs, with specialized operators to ensure syntactic correctness of offspring. A detailed history is available in *Fogel*'s 2006 book [43.51].

In the 1980s, inspired by the success of genetic algorithms (GAs) and learning classifier systems (LCSs), several authors experimented with hierarchically structured and program-like representations. *Smith* [43.52] proposed a representation of a variable-length list of rules which could be used for program-like behavior such as maze navigation and poker. *Cramer* [43.53] was the first to use a tree-structured representation and appropriate operators. With a simple proof of concept, it successfully evolved a multiplication function in a simple custom language. *Schmidhuber* [43.54] describes a GP system with the possibility of Turing completeness, though the focus is on meta-learning aspects. *Fujiki* and *Dickinson* [43.55] generated Lisp code for the prisoner's dilemma, *Bickel* and *Bickel* [43.56] used a GA to create variable-length lists of rules, each of which had a tree structure. An artificial life approach using machine-code genomes was used by *Ray* [43.57]. All of these would likely be regarded as on-topic in a modern GP conference.

However, the founding of the modern field of GP, and the invention of what is now called *standard* GP, are credited to *Koza* [43.19]. In addition to the abstract syntax tree notation (Sect. 43.3.3), the key innovations were subtree crossover (Sect. 43.3.3) and the description and set-up of many test problems. In this and

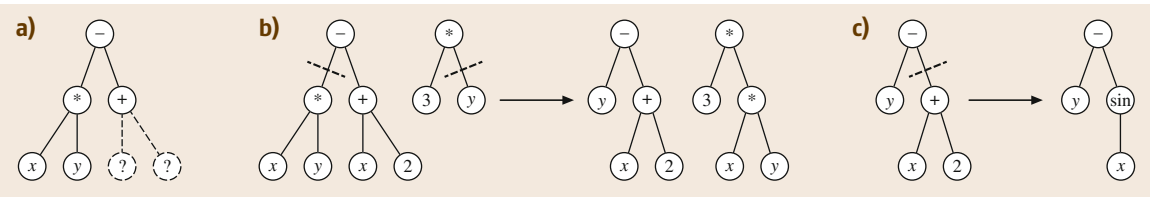

Fig. 43.2a–c The StdGP representation is an abstract syntax tree. The expression that will be evaluated in the *second tree from left* is, in inorder notation, $(x * y) - (x + 2)$. In preorder, or the notation of Lisp-style S-expressions, it is $(- (* x\ y)\ (+ x\ 2))$. GP presumes that the variables x and y will be already bound to some value in the execution environment when the expression is evaluated. It also presumes that the operations $*$ and $-$, etc. are also defined. Note that, all interior tree nodes are effectively operators in some computational language. In standard GP parlance, these operators are called *functions* and the leaf tree nodes which accept no arguments and typically represent variables bound to data values from the problem domain are referred to as *terminals*

later research [43.10, 58, 59] symbolic regression of synthetic data and real-world time series, Boolean problems, and simple robot control problems such as the lawnmower problem and the artificial ant with Santa Fe trail were introduced as benchmarks and solved successfully for the first time, demonstrating that GP was a potentially powerful and general-purpose method capable of solving machine learning-style problems albeit conventional academic versions of them. Mutation was minimized in order to make it clear that GP was different from random search. GP took on its modern form in the years following Koza's 1992 book: many researchers took up work in the field, new types of GP were developed (Sect. 43.3), successful applications appeared (Sect. 43.4), key research topics were identified (Sect. 43.5), further books were written, and conferences and journals were established (Sect. 43.6).

Another important milestone in the history of GP was the 2004 establishment of the *Humies*, the awards for human-competitive results produced by EC methods. The entries are judged for matching or exceeding human-produced solutions to the same or similar problems, and for criteria such as patentability and publishability. The impressive list of human-competitive results [43.60] again helps to demonstrate to researchers and clients outside the field of GP that it is powerful and general purpose.

43.3 Taxonomy of AI and GP

In this section, we present a taxonomy which firstly places GP in the context of the broader fields of EC, ML, and artificial intelligence (AI). It then classifies GP techniques according to their representations and their population models (Fig. 43.3).

43.3.1 Placing GP in an AI Context

GP is a type of EC, which is a type of ML, which is itself a subset of the broader field of AI (Fig. 43.3). *Carbonell* et al. [43.61] classify ML techniques according to the underlying learning strategy, which may be rote learning, learning from instruction, learning by analogy, learning from examples, and learning from observation and discovery. In this classification, EC and GP fit in the *learning from examples* category, in that an (individual, fitness) pair is an example drawn from the search space together with its evaluation.

It is also useful to see GP as a subset of another field, AP. The term *automatic programming* seems to have had different meanings at different times, from automated card punching, to compilation, to template-driven source generation, then generation techniques such as universal modeling language (UML), to the ambitious aim of creating software directly from a natural-language English specification [43.62]. We interpret AP to mean creating software by specifying *what to do* rather than *how to do it* [43.47]. GP clearly fits into this category. Other nonevolutionary techniques also do so, for example inductive programming (IP). The main difference between GP and IP is that typically IP works only with programs which are known to be correct, achieving this using inductive methods over the spec-

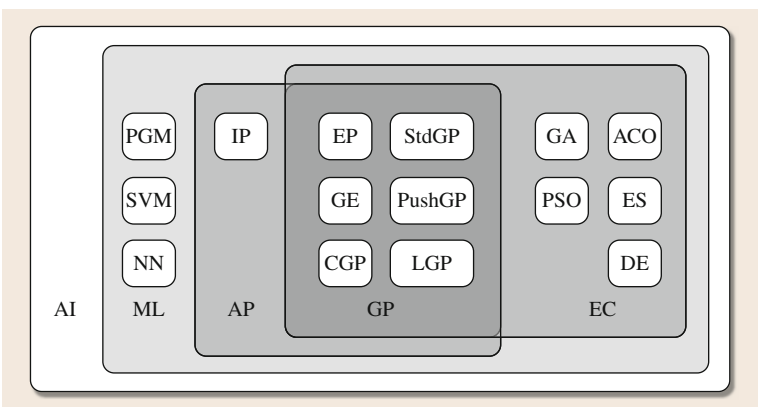

Fig. 43.3 A taxonomy of AI, EC, and GP

ifications, [43.63]. In contrast, GP is concerned mostly with programs which are syntactically correct, but behaviorally suboptimal.

43.3.2 Taxonomy of GP

It is traditional to divide EC into four main subfields: evolution strategies (ES) [43.64, 65], evolutionary programming (EP) [43.50], GAs [43.66], and GP. In this view, ES is chiefly characterized by real-valued optimization and self-adaptation of algorithm parameters; EP by a finite-state machine representation (later generalized) and the absence of crossover; GA by the bitstring representation; and GP by the abstract syntax tree representation. While historically useful, this classification is not exhaustive: in particular it does not provide a home for the many alternative GP representations which now exist. It also separates EP and GP, though they are both concerned with evolving programs. We prefer to use the term GP in a general sense to refer to all types of EC which evolve programs. We use the term *standard GP* (StdGP) to mean Koza-style GP with a tree representation. With this view, StdGP and EP are types of GP, as are several others discussed below. In the following, we classify GP algorithms according to their *representation* and according to their *population model*.

43.3.3 Representations

Throughout EC, it is useful to contrast *direct* and *indirect* representations. Standard GP is direct, in that the genome (the object created and modified by the genetic operators) serves directly as an executable program. Some other GP representations are indirect, meaning that the genome must be decoded or translated in some way to give an executable program. An example is grammatical evolution (GE, see below), where the genome is an integer array which is used to generate a program. Indirect representations have the advantage that they may allow an easier definition of the genetic operators, since they may allow the genome to exist in a rather simpler space than that of executable programs. Indirect representations also imitate somewhat more closely the mechanism found in nature, a mapping from DNA (deoxyribonucleic acid) to RNA (ribonucleic acid) to mRNA (messenger RNA) to codons to proteins and finally to cells. The choice between direct and indirect representations also affects the structure of the fitness landscape (Sect. 43.5.2). In the following, we present a nonexhaustive selection of the main representations used in GP, in each case describing initialization and the two key operators: mutation, and crossover.

Standard GP

In Standard GP (StdGP), the representation is an abstract syntax tree, or can be seen as a Lisp-style S-expression. All nodes are functions and all arguments are the same type. A function accepts zero or more arguments and returns a single value. Trees can be initialized by recursive random growth starting from a null node. StdGP uses parameterized initialization methods that diversify the size and structure of initial trees. Figure 43.2a shows a tree in the process of initialization.

Trees can be crossed over by cutting and swapping the subtrees rooted at randomly chosen nodes, as shown in Fig. 43.2b. They can be mutated by cutting and regrowing from the subtrees of randomly chosen nodes, as shown in Fig. 43.2c. Another mutation operator, HVL-Prime, is shown later in Fig. 43.11. Note that crossover or mutation creates an offspring of potentially different size and structure, but the offspring remains syntactically valid for evaluation. With these variations, a tree could theoretically grow to infinite size or height. To circumvent this, as a practicality, a hard parameterized threshold for size or height or some other threshold is used. Violations to the threshold are typically rejected. Bias may also be applied in the randomized selection of crossed-over subtree roots. A common variation of StdGP is *strongly typed GP* (STGP) [43.67, 68], which supports functions accepting arguments and returning values of specific types by means of specialized mutation and crossover operations that respect these types.

Executable Graph Representations

A natural generalization of the executable tree representation of StdGP is the executable graph. Neural networks can be seen as executable graphs in which each node calculates a weighted sum of its inputs and outputs the result after a fixed shaping function such as tanh(). *Parallel and distributed GP* (PDGP) [43.69] is more closely akin to StdGP in that nodes calculate different functions, depending on their labels, and do not perform a weighted sum. It also allows the topology of the graph to vary, unlike the typical neural network. *Cartesian GP* (CGP) [43.70] uses an integer-array genome and a mapping process to produce the graph. Each block of three integer genes codes for a single node in the graph, specifying the indices of

its inputs and the function to be executed by the node (Fig. 43.4).

Neuro-evolution of augmenting topologies (NEAT) [43.71] again allows the topology to vary, and allows nodes to be labelled by the functions they perform, but in this case each node does perform a weighted sum of its inputs. Each of these representations uses different operators. For example, CGP uses simple array-oriented (GA-style) initialization, crossover, and mutation operators (subject to some customizations).

Finite-State Machine Representations

Some GP representations use graphs in a different way: the model of computation is the finite-state machine rather than the executable functional graph (Fig. 43.5). The original incarnation of *evolutionary programming* (EP) [43.72] is an example. In a typical implementation [43.72], five types of mutation are used: adding and deleting states, changing the initial state, changing the output symbol attached to edges, and changing the edges themselves. In this implementation, crossover is not used.

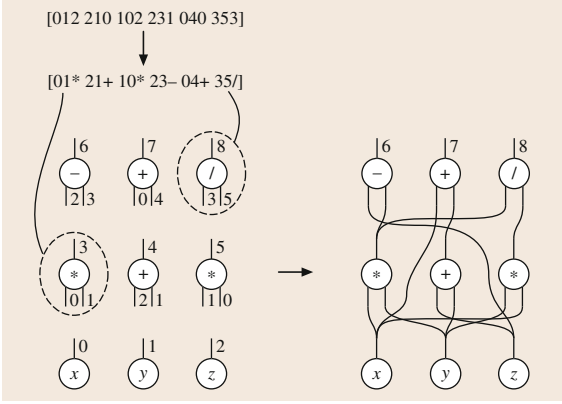

Fig. 43.4 Cartesian GP. An integer-array genome is divided into blocks: in each block the last integer specifies a function (*top-left*). Then one node is created for each input variable (x, y, z) and for each genome block. Nodes are arranged in a grid and outputs are indexed sequentially (*bottom-left*). The *first elements in each block* specify the indices of the incoming links. The *final graph* is created by connecting each node input to the node output with the same integer label (*right*). Dataflow in the graph is *bottom to top*. Multiple outputs can be read from the topmost layer of nodes. In this example node 6 outputs $xy-z+y$, node 7 outputs $x+z+y$, and node 8 outputs xy/xy

Grammatical GP

In *grammatical GP* [43.73], the context-free grammar (CFG) is the defining component of the representation. In the most common approach, search takes place in the space defined by a fixed nondeterministic CFG. The aim is to find a good program in that space. Often the CFG defines a useful subset of a programming language such as Lisp, C, or Python. Programs derived from the CFG can then be compiled or interpreted using either standard or special-purpose software. There are several advantages to using a CFG. It allows convenient definition of multiple data-types which are automatically respected by the crossover and mutation operators. It can introduce domain knowledge into the problem representation. For example, if it is known that good programs will consist of a conditional statement inside a loop, it is easy to express this knowledge using a grammar. The grammar can restrict the ways in which program expressions are combined, for example making the system aware of physical units in *dimensionally aware* GP [43.74, 75]. A grammatical GP system can conveniently be applied to new domains, or can incorporate new domain knowledge, through updates to the grammar rather than large-scale reprogramming.

In one early system [43.76], the derivation tree is used as the genome: initial individuals' genomes are randomly generated according to the rules of the grammar. Mutation works by randomly generating a new subtree starting from a randomly chosen internal node in the derivation tree. Crossover is constrained to exchange subtrees whose roots are identical. In this way, new individuals are guaranteed to be valid derivation trees. The executable program is then created from the genome by reading the leaves left to right. A later system, *grammatical evolution* (GEs) [43.77] instead uses an integer-array genome. Initialization, mutation and crossover are defined as simple GA-style array operations. The genome is mapped to an output program by using the successive integers of the genome to choose

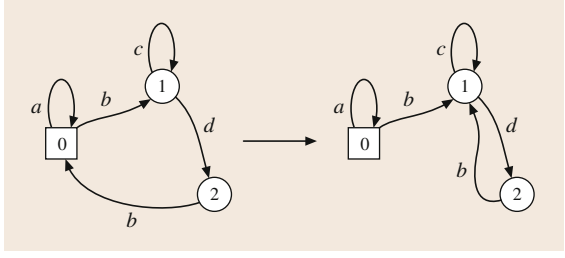

Fig. 43.5 EP representation: finite-state machine. In this example, a mutation changes a state transition

among the applicable production choices at each step of the derivation process. Figure 43.6 shows a simple grammar, integer genome, derivation process, and derivation tree. At each step of the derivation process, the left-most nonterminal in the derivation is rewritten. The next integer gene is used to determine, using the *mod rule*, which of the possible productions is chosen. The output program is the final step of the derivation tree.

Although successful and widely used, GE has also been criticized for the disruptive effects of its operators with respect to preserving the modular functionality of parents. Another system, *tree adjoining grammar-guided genetic programming* (TAG3P) has also been used successfully [43.78]. Instead of a string-rewriting CFG, TAG3P uses the tree-rewriting *tree adjoining grammars*. The representation has the advantage, relative to GE, that individuals are valid programs at every step of the derivation process. TAGs also have some context-sensitive properties [43.78]. However, it is a more complex representation.

Another common alternative approach, surveyed by *Shan* et al. [43.79], uses probabilistic models over grammar-defined spaces, rather than direct evolutionary search.

Linear GP

In *Linear GP* (LGP), the program is a list of instructions to be interpreted sequentially. In order to achieve complex functionality, a set of registers acting as state or memory are used. Instructions can read from or write to the registers. Several registers, which may be read-only, are initialized with the values of the input variables. One register is designated as the output: its value at the end of the program is taken as the result of the program. Since a register can be read multiple times after writing, an LGP program can be seen as having a graph structure. A typical implementation is that of [43.80]. It uses instructions of three registers each, which typically calculate a new value as an arithmetic function of some registers and/or constants, and assign it to a register (Fig. 43.7).

It also allows conditional statements and looping. It explicitly recognizes the possibility of nonfunctioning code, or *introns*. Since there are no syntactic constraints on how multiple instructions may be composed together, initialization can be as simple as the random generation of a list of valid instructions. Mutation can change a single instruction to a newly generated instruction, or change just a single element of an instruction. Crossover can be performed over the two parents' list structures, respecting instruction boundaries.

Stack-Based GP

A variant of linear GP avoids the need for registers by adding a stack. The program is again a list of instructions, each now represented by a single label. In a simple arithmetic implementation, the label may be one of the input variables (x_i), a numerical constant, or a function ($*, +$, etc.). If it is a variable or constant, the instruction is executed by pushing the value onto the stack. If a function, it is executed by popping the required number of operands from the stack, executing the function on them, and pushing the result back on. The result of the program is the value at the top of

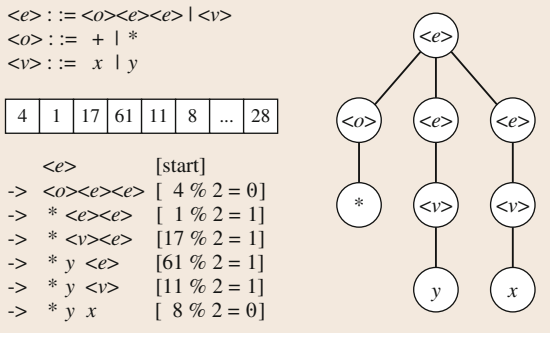

Fig. 43.6 GE representation. The grammar (*top-left*) consists of several rules. The genome (*center-left*) is a variable-length list of integers. At each step of the derivation process (*bottom-left*), the left-most nonterminal is rewritten as specified by a gene. The resulting derivation tree is shown on the *right*: reading just the leaves gives the derived program

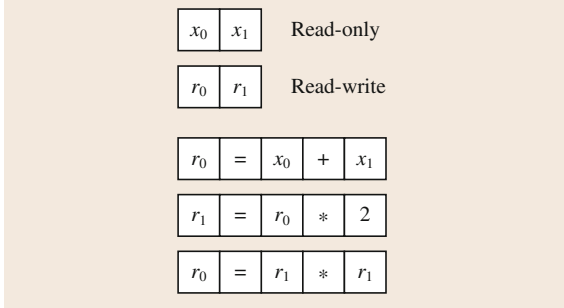

Fig. 43.7 Linear GP representation. This implementation has four registers in total (*top*). The representation is a list of register-oriented instructions (*bottom*). In this example program of three instructions, r_0 is the output register, and the formula $4(x_0 + x_1)^2$ is calculated

the stack after all instructions have been executed. With the stipulation that stack-popping instructions become no-ops when the stack is empty, one can again implement initialization, mutation, and crossover as simple list-based operations [43.81]. One can also constrain the operations to work on what are effectively subtrees, so that stack-based GP becomes effectively equivalent to a reverse Polish notation implementation of standard GP [43.82]. A more sophisticated type of stack-based GP is *PushGP* [43.83], in which multiple stacks are used. Each stack is used for values of a different type, such as integer, boolean, and float. When a function requires multiple operands of different types, they are taken as required from the appropriate stacks. With the addition of an *exec* stack which stores the program code itself, and the *code* stack which stores items of code, both of which may be both read and written, PushGP gains the ability to evolve programs with self-modification, modularity, control structures, and even self-reproduction.

Low-Level Programming

Finally, several authors have evolved programs directly in real-world low-level programming languages. *Schulte* et al. [43.84] automatically repaired programs written in Java byte code and in x86 assembly. *Orlov* and *Sipper* [43.85] evolved programs such as trail navigation and image classification de novo in Java byte code. This work made use of a specialized crossover operator which performed automated checks for compatibility of the parent programs' stack and control flow state. *Nordin* [43.86] proposed a machine-code representation for GP. Programs consist of lists of low-level register-oriented instructions which execute directly, rather than in a virtual machine or interpreter. The result is a massive speed-up in execution.

43.3.4 Population Models

It is also useful to classify GP methods according to their population models. In general the population model and the representation can vary independently, and in fact all of the following population can be applied with any EC representation including bitstrings and real-valued vectors, as well as with GP representations.

The simplest possible model, *hill-climbing*, uses just one individual at a time [43.87]. At each iteration, offspring are created until one of them is more highly fit than the current individual, which it then replaces. If at any iteration it becomes impossible to find an improvement, the algorithm has *climbed the hill*, i. e. reached a local optimum, and stops. It is common to use a random restart in this case. The hill-climbing model can be used in combination with any representation. Note that it does not use crossover. Variants include ES-style (μ, λ) or $(\mu + \lambda)$ schemes, in which multiple parents each give rise to multiple offspring by mutation.

The most common model is an *evolving population*. Here a large number of individuals (from tens to many thousands) exist in parallel, with new generations being created by crossover and mutation among selected individuals. Variants include the steady-state and the generational models. They differ only in that the steady-state model generates one or a few new individuals at a time, adds them to the existing population and removes some old or weak individuals; whereas the generational model generates an entirely new population all at once and discards the old one.

The *island model* is a further addition, in which multiple populations all evolve in parallel, with infrequent migration between them [43.88].

In *coevolutionary* models, the fitness of an individual cannot be calculated in an endogenous way. Instead it depends on the individual's relationship to other individuals in the population. A typical example is in game-playing applications such as checkers, where the best way to evaluate an individual is to allow it to play against other individuals. Coevolution can also use fitness defined in terms of an individual's relationship to individuals in a population of a different type. A good example is the work of [43.89], which uses a type of *predator–prey* relationship between populations of programs and populations of test cases. The test cases (*predators*) evolve to find bugs in the programs; the programs (*prey*) evolve to fix the bugs being tested for by the test suites.

Another group of highly biologically inspired population models are those of *swarm intelligence*. Here the primary method of learning is not the creation of new individuals by inheritance. Instead, each individual generally lives for the length of the run, but *moves about* in the search space with reference to other individuals and their current fitness values. For example, in particle swarm optimization (PSO) individuals tend to move toward the global best and toward the best point in their own history, but tend to avoid moving too close to other individuals. Although PSO and related methods such as differential evolution (DE) are best applied in real-valued optimization, their population models and operators can be abstracted and applied in GP methods also [43.90, 91].

Finally, we come to *estimation of distribution algorithms* (EDAs). Here the idea is to create a population, select a subsample of the best individuals, model that subsample using a distribution, and then create a new population by sampling the distribution. This approach is particularly common in grammar-based GP [43.73], though it is also used with other representations [43.92–94]. The modeling-sampling process could be regarded as a type of whole-population crossover. Alternatively one can view EDAs as being quite far from the biological inspiration of most EC, and in a sense they bridge the gap between EC and statistical ML.

43.4 Uses of GP

Our introduction (Sect. 43.1) has touched on a wide array of domains in which GP has been applied. In this section, we give more detail on just a few of these.

43.4.1 Symbolic Regression

Symbolic regression is one of the most common tasks for which GP is used [43.19, 95, 96]. It is used as a component in techniques like data modeling, clustering, and classification, for example in the modeling application outlined in Sect. 43.4.2. It is named after techniques such as linear or quadratic regression, and can be seen as a generalization of them. Unlike those techniques it does not require a priori specification of the model. The goal is to find a function in symbolic form which models a data set. A typical symbolic regression is implemented as follows.

It begins with a dataset which is to be regressed, in the form of a numerical matrix (Fig. 43.8, left). Each row i is a data-point consisting of some input (explanatory) variables $\boldsymbol{x}_i$ and an output (response) variable y_i to be modeled. The goal is to produce a function $f(\boldsymbol{x})$ which models the relationship between $\boldsymbol{x}$ and y as closely as possible. Figure 43.8 (right) plots the existing data and one possible function f.

Typically StdGP is used, with a numerical *language* which includes arithmetic operators, functions like sinusoids and exponentials, numerical constants, and the input variables of the dataset. The internal nodes of each StdGP abstract syntax tree will be operators and functions, and the leaf nodes will be constants and variables.

To calculate the fitness of each model, the explanatory variables of the model are bound to their values at each of the training points $\boldsymbol{x}_i$ in turn. The model is executed, and the output $f(\boldsymbol{x}_i)$ is the model's predicted response. This value $\hat{y}_i$ is then compared to the response of the training point y_i. The error can be visualized as the dotted lines in Fig. 43.8 (right). Fitness is usually defined as the root-mean-square error of the model's outputs versus the training data. In this formulation, therefore, fitness is to be minimized

$$\text{fitness}(f) = \sqrt{\frac{\sum_{i=1}^{n}(f(\boldsymbol{x}_i) - y_i)^2}{n}} \,.$$

Over the course of evolution, the population moves toward better and better models f of the training data. After the run, a testing data set is used to confirm that the model is capable of generalization to unseen data.

43.4.2 Machine Learning

Like other ML methods, GP is successful in quantitative domains where data is available for learning and both approximate solutions and incremental improvements are valued. In modeling or supervised learning, GP is preferable to other ML methods in circumstances where the form of the solution model is unknown a priori because it is capable of searching among possible forms for the model. Symbolic regression can be used as an approach to classification, regression modeling, and clustering. It can also be used to automatically extract influential features, since it is able to pare down the feature set it is given at initialization. GP-derived classifiers have been integrated into ensemble

x	y
0.1	0.3
0.2	0.6
0.3	0.5
0.4	0.7
0.5	0.5
.	.
.	.

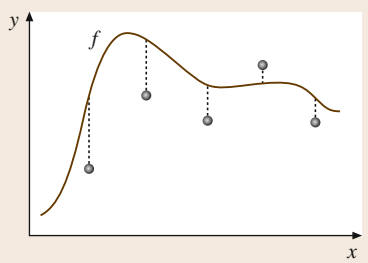

Fig. 43.8 Symbolic regression: a matrix of data (*left*) is to be modeled by a function. It is plotted as dots in the figure on the right. A candidate function f (*solid line*) can be plotted, and its errors (*dotted lines*) can be visualized

learning approaches and GP has been used in reinforcement learning (RL) contexts. Figure 43.9 shows GP as a means of ML which allows it to address problems such as planning, forecasting, pattern recognition, and modeling.

For the sensory evaluation problem described in [43.97], the authors use GP as the anchor of a ML framework (Fig. 43.10). A panel of assessors provides *liking scores* for many different flavors. Each flavor consists of a mixture of ingredients in different proportions. The goals are to discover the dependency of a liking score on the concentration levels of flavors' ingredients, identifying ingredients that drive liking, segmenting the panel into groups with similar liking preferences and optimizing flavors to maximize liking per group. The framework uses symbolic regression and ensemble methods to generate multiple diverse explanations of liking scores, with confidence information. It uses statistical techniques to extrapolate from the genetically evolved model ensembles to unobserved regions of the flavor space. It also segments the assessors into groups which either have the same propensity to like flavors, or whose liking is driven by the same ingredients.

Sensory evaluation data is very sparse and there is large variation among the responses of different assessors. A Pareto-GP algorithm (which uses multi-objective techniques to maximise model accuracy and minimise model complexity; [43.98]) was therefore used to evolve an ensemble of models for each assessor and to use this ensemble as a source of robust variable importance estimation. The frequency of variable occurrences in the models of the ensemble was interpreted as information about the ingredients that drive the liking of an assessor. Model ensembles with the same dominance of variable occurrences, and which demonstrate similar effects when the important variables are varied, were grouped together to identify assessors who are driven by the same ingredient set and in the same direction. Varying the input values of the important variables, while using the model ensembles of these panel segments, provided a means of conducting focused sensitivity analysis. Subsequently, the same model ensembles when clustered constitute the *black box* which is used by an evolutionary algorithm in its optimization of flavors that are well liked by assessors who are driven by the same ingredient.

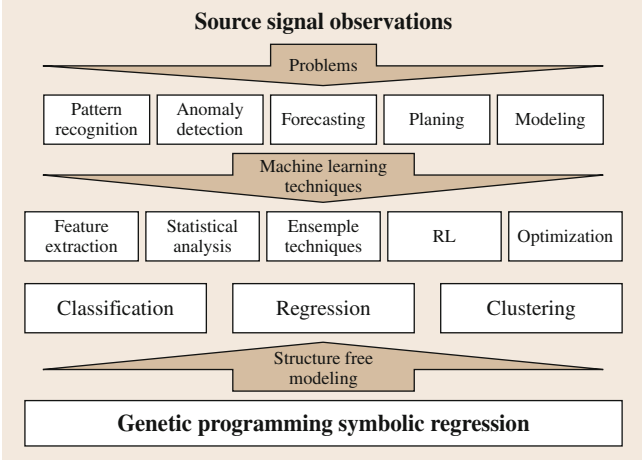

Fig. 43.9 GP as a component in ML. Symbolic regression can be used as an approach to many ML tasks, and integrated with other ML techniques

43.4.3 Software Engineering

At least three areas of software engineering have been tackled with remarkable success by GP: bug-fixing [43.99], parallelization [43.100, 101], and optimization [43.102–104]. These three areas are very different in their aims, scope, and methods; however, they all need to deal with two key problems in this domain: the very large and unconstrained search space, and the problem of program correctness. They therefore do not aim to evolve new functionality from scratch, but instead use existing code as material to be transformed in some way; and they either guarantee correctness of the evolved programs as a result of their representations, or take advantage of existing test suites in order to provide strong evidence of correctness.

Le Goues et al. [43.99] show that automatically fixing software bugs is a problem within the reach of GP. They describe a system called *GenProg*. It operates on C source code taken from open-source projects. It works by forming an abstract syntax tree from the original source code. The initial population is seeded with variations of the original. Mutations and crossover are constrained to copy or delete complete lines of code, rather than editing subexpressions, and they are constrained to alter only lines which are exercised by the failing test cases. This helps to reduce the search space size. The original test suites are used to give confidence that the program variations have not lost their original functionality. Fixes for several real-world bugs are produced, quickly and with high certainty of success, including bugs in HTTP servers, Unix utilities, and a media player. The fixes can be automatically processed to produce minimal patches. Best of all, the fixes

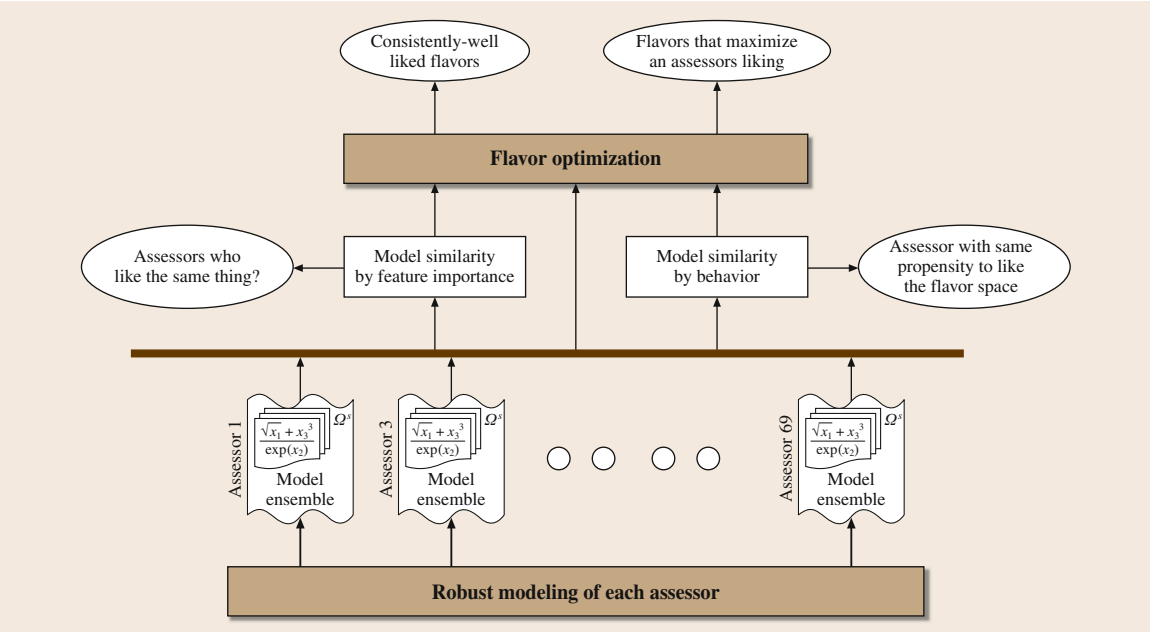

Fig. 43.10 GP symbolic regression is unique and useful as an ML technique because it obviates the need to define the structure of a model prior to training. Here, it is used to form a personalized ensemble model for each assessor in a flavor evaluation panel

are demonstrated to be rather robust: some even generalize to fixing related bugs which were not explicitly encoded in the test suite.

Ryan [43.100] describes a system, *Paragen*, which automatically rewrites serial Fortran programs to parallel versions. In *Paragen I*, the programs are directly varied by the genetic operators, and automated tests are used to reward the preservation of the program's original semantics. The work of *Williams* [43.101] was in some ways similar to Paragen I. In *Paragen II*, correctness of the new programs is instead guaranteed, using a different approach. The programs to be evolved are sequences of transformations defined over the original serial code. Each transformation is known to preserve semantics. Some transformations however directly transform serial operations to parallel, while other transformations merely enable the first type.

A third goal of software engineering is optimization of existing code. *White* et al. [43.104] tackle this task using a multiobjective optimization method. Again, an existing program is used as a starting point, and the aim is to evolve a semantically equivalent one with improved characteristics, such as reduced memory usage, execution time, or power consumption. The system is capable of finding *nonobvious* optimizations, i. e. ones which cannot be found by optimizing compilers. A population of test cases is coevolved with the population of programs. *Stephenson* et al. [43.102, 103] in the Meta Optimization project improve program execution speed by using GP to refine priority functions within the compiler. The compiler generates better code which executes faster across the input range of one program and across the program range of a benchmark set.

A survey of the broader field of *search-based software engineering* is given by *Harman* [43.105].

43.4.4 Design

GP has been successfully used in several areas of design. This includes both engineering design, where the aim is to design some hardware or software system to carry out a well-defined task, and aesthetic design, where the aim is to produce art objects with subjective qualities.

Engineering Design

One of the first examples of GP design was the synthesis of analog electrical circuits by *Koza* et al. [43.106]. This work addressed the problem of automatically cre-

ating circuits to perform tasks such as a filter or an amplifier. Eight types of circuit were automatically created, each having certain requirements, such as outputting an amplified copy of the input, and low distortion. These functions were used to define fitness. A complex GP representation was used, with both STGP (Sect. 43.3.3) and ADFs (Sect. 43.5.3). Execution of the evolved program began with a trivial *embryonic circuit*. GP program nodes, when executed, performed actions such as altering the circuit topology or creating a new component. These nodes were parameterized with numerical parameters, also under GP control, which could be created by more typical arithmetic GP subtrees. The evolved circuits solved significant problems to a human-competitive standard though they were not fabricated.

Another significant success story was the space-going antenna evolved by *Hornby* et al. [43.27] for the NASA (National Aeronautics and Space Administration) Space Technology 5 spacecraft. The task was to design an antenna with certain beamwidth and bandwidth requirements, which could be tested in simulation (thus providing a natural fitness function). GP was used to reduce reliance on human labor and limitations on complexity, and to explore areas of the search space which would be rejected as not worthy of exploration by human designers. Both a GA and a GP representation were used, producing quite similar results. The GP representation was in some ways similar to a 3-D turtle graphics system. Commands included *forward* which moved the turtle forward, creating a wire component, and *rotate-x* which changed orientation. Branching of the antenna arms was allowed with special markers similar to those used in turtle graphics programs. The program composed of these primitives, when run, created a wire structure, which was rotated and copied four times to produce a symmetric result for simulation and evaluation.

Aesthetic Design

There have also been successes in the fields of graphical art, 3-D aesthetic design, and music. Given the aesthetic nature of these fields, GP fitness is often replaced by an interactive approach where the user performs *direct selection* on the population. This approach dates back to *Dawkins'* seminal *Biomorphs* [43.107] and has been used in other forms of EC also [43.108]. Early successes were those of *Todd* and *Latham* [43.109], who created pseudo-organic forms, and *Sims* [43.35] who created abstract art. An overview of evolutionary art is provided by *Lewis* [43.110].

A key aim throughout aesthetic design is to avoid the many random-seeming designs which tend to be created by typical representations. For example, a naive representation for music might encode each quarter-note as an integer in a genome whose length is the length of the eventual piece. Such a representation will be capable of representing some good pieces of music, but it will have several significant problems. The vast majority of pieces will be very poor and random sounding. Small mutations will tend to gradually degrade pieces, rather than causing large-scale and semantically sensible transformations [43.111].

As a result, many authors have tried to use representations which take advantage of forms of *reuse*. Although reuse is also an aim in nonaesthetic GP (Sect. 43.5.3), the hypothesis that good solutions will tend to involve reuse, even on new, unknown problems, is more easily motivated in the context of aesthetic design.

In one strand of research, the time or space to be occupied by the work is predefined, and divided into a grid of 1, 2, or 3 dimensions. A GP function of 1, 2 or 3 arguments is then evolved, and applied to each point in the grid with the coordinates of the point passed as arguments to the function. The result is that the function is reused many times, and all parts of the work are felt to be coherent. The earliest example of such work was that of *Sims* [43.35], who created fascinating graphical art (a 2-D grid) and some animations (a 3-D grid of two spatial dimensions and 1 time dimension). The paradigm was later brought to a high degree of artistry by *Hart* [43.112]. The same generative idea, now with a 1-D grid representing time, was used by *Hoover* et al. [43.113], *Shao* et al. [43.114] and *McDermott* and *O'Reilly* [43.115] to produce music as a function of time, and with a 3-D grid by *Clune* and *Lipson* [43.116] to produce 3-D sculptures.

Other successful work has used different approaches to reuse. *L-systems* are grammars in which symbols are recursively expanded in parallel: after several expansions (a *growth process*), the string will by highly patterned, with multiple copies of some substrings. Interpreting this string as a program can then yield highly patterned graphics [43.117], artificial creatures [43.118], and music [43.119]. Grammars have also been used in 3-D and architectural design, both in a modified L-system form [43.36] and in the standard GE form [43.120]. The *Ossia* system of *Dahlstedt* [43.37] uses GP trees with recursive pointers to impose reuse and a natural, *gestural* quality on short pieces of art music.

43.5 Research Topics

Many research topics of interest to GP practitioners are also of broader interest. For example, the self-adaptation of algorithm parameters is a topic of interest throughout EC. We have chosen to focus on four research topics of specific interest in GP: bloat, GP theory, modularity, and open-ended evolution.

43.5.1 Bloat

Most GP-type problems naturally require variable-length representations. It might be expected that selection pressure would effectively guide the population toward program sizes appropriate to the problem, and indeed this is sometimes the case. However, it has been observed that for many different representations [43.121] and problems, programs grow over time *without* apparent fitness improvements. This phenomenon is called *bloat*. Since the time complexity for the evaluation of a GP program is generally proportional to its size, this greatly slows the GP run down. There are also other drawbacks. The eventual solution may be so large and complex that is unreadable, negating a key advantage of symbolic methods like GP. Overly large programs tend to generalize less well than parsimonious ones. Bloat may negatively impact the rate of fitness improvement. Since bloat is a significant obstacle to successful GP, it is an important topic of research, with differing viewpoints both on the causes of bloat and the best solutions.

The competing theories of the causes of bloat are summarized by *Luke* and *Panait* [43.122] and *Silva* et al. [43.123]. A fundamental idea is that adding material rather than removing material from a GP tree is more likely to lead to a fitness improvement. The *hitchhiking* theory is that noneffective code is carried along by virtue of being attached to useful code. *Defense against crossover* suggests that large amounts of noneffective code give a selection advantage later in GP runs when crossover is likely to highly destructive of good, fragile programs. *Removal bias* is the idea that it is harder for GP operators to remove exactly the right (i. e., noneffective) code than it is to add more. The *fitness causes bloat* theory suggests that fitness-neutral changes tend to increase program size just because there are many more programs with the same functionality at larger sizes than at smaller [43.124]. The *modification point depth* theory suggests that children formed by tree crossover at deep crossover points are likely to have fitness similar to their parents and thus more likely to survive than the more radically different children formed at shallow crossover points. Because larger trees have more very deep potential crossover points, there is a selection pressure toward growth. Finally, the *crossover bias* theory [43.125] suggests that after many crossovers, a population will tend toward a limiting distribution of tree sizes [43.126] such that small trees are more common than large ones – note that this is the opposite of the effect that might be expected as the basis of a theory of bloat. However, when selection is considered, the majority of the small programs cannot compete with the larger ones, and so the distribution is now skewed in favour of larger programs.

Many different solutions to the problem of bloat have been proposed, many with some success. One simple method is *depth limiting*, imposing a fixed limit on the tree depth that can be produced by the variation operators [43.19].

Another simple but effective method is *Tarpeian bloat control* [43.127]. Individuals which are larger than average receive, with a certain probability, a constant punitively bad fitness. The advantage is that these individuals are not evaluated, and so a huge amount of time can be saved and devoted to running more generations (as in [43.122]). The Tarpeian method does allow the population to grow beyond its initial size, since the punishment is only applied to a proportion of individuals – typically around 1 in 3. This value can also be set adaptively [43.127].

The *parsimony pressure* method evaluates all individuals, but imposes a fitness penalty on overly large individuals. This assumes that fitness is commensurable with size: the magnitude of the punishment establishes a *de facto exchange rate* between the two. *Luke* and *Panait* [43.122] found that parsimony pressure was effective across problems and across a wide range of exchange rates.

The choice of an exchange rate can be avoided using multiobjective methods, such as Pareto-GP [43.128], where one of the objectives is fitness and the other program length or complexity. The correct definition for complexity in this context is itself an interesting research topic [43.96, 129]. Alternatively, the pressure against bloat can be moved from the fitness evaluation phase to the the selection phase of the algorithm, using the *double tournament* method [43.122]. Here individuals must compete in one fitness-based tournament and one size-based one. Another approach

is to incorporate tree size directly into fitness evaluation using a minimum description length principle [43.130].

Another technique is called *operator length equalization*. A histogram of program sizes is maintained throughout the run and is used to set the population's capacity for programs of different sizes. A newly created program which would cause the population's capacity to be exceeded is rejected, unless exceptionally fit. A *mutation-based* variation of the method instead mutates the overly large individuals using directed mutation to become smaller or larger as needed.

Some authors have argued that the choice of GP representation can avoid the issue of bloat [43.131]. Some aim to avoid the problem of bloat by speeding up fitness evaluation [43.82, 132] or avoiding wasted effort in evaluation [43.133, 134]. Sometimes GP techniques are introduced with other motivations but have the side-effect of reducing bloat [43.135].

In summary, researchers including *Luke* and *Panait* [43.122], *Poli* et al. [43.127], *Miller* [43.131], and *Silva* et al. [43.123] have effectively *declared victory* in the fight against bloat. However, their techniques have not yet become standard for new GP research and benchmark experiments.

43.5.2 GP Theory

Theoretical research in GP seeks to answer a variety of questions, for example: What are the drivers of population fitness convergence? How does the behavior of an operator influence the progress of the algorithm? How does the combination of different algorithmic mechanisms steer GP toward fitter solutions? What mechanisms cause bloat to arise? What problems are difficult for GP? How diverse is a GP population? Theoretical methodologies are based in mathematics and exploit formalisms, theorems, and proofs for rigor. While GP may appear simple, beyond its stochastic nature which it shares with all other evolutionary algorithms, its variety of representations each impose specific requirements for theoretical treatment. All GP representations share two common traits which greatly contribute to the difficulty it poses for theoretical analysis. First, the representations have no fixed size, implying a complex search space. Second, GP representations do not imply that parents will be equal in size and shape. While crossover accommodates this lack of synchronization, it generally allows the exchange of content from *anywhere* in one parent to *anywhere* in the other parent's tree. This implies combinatorial outcomes and *likes not switching with likes*. This functionality contributes to complicated algorithmic behavior which is challenging to analyze.

Here, we select several influential methods of theoretical analysis and very briefly describe them and their results: schema-based analysis, Markov chain modeling, runtime complexity, and problem difficulty. We also introduce the No Free Lunch Theorem and describe its implications for GP.

Schema-Based Analysis

In schema-based analysis, the search space is conceptually partitioned into hyperplanes (also known as schemas) which represent sets of partial solutions. There are numerous ways to do this and, as a consequence, multiple schema definitions have been proposed [43.136–139]. The fitness of a schema is estimated as the average fitness of all programs in the sample of its hyperplane, given a population. The processes of fitness-based selection and crossover are formalized in a recurrence equation which describes the expected number of programs sampling a schema from the current population to the next. Exact formulations have been derived for most types of crossover [43.140, 141]. These alternatively depend on making explicit the effects and the mechanisms of schema creation. This leads to insight; however, tracking schema equations in actual GP population dynamics is infeasible. Also, while schema theorems predict changes from one generation to the next, they cannot predict further into the future to predict the long-term dynamics that GP practitioners care about.

Markov Chain Analysis

Markov chain models are one means of describing such long-term GP dynamics. They take advantage of the Markovian property observed in a GP algorithm: the composition of one generation's population relies only upon that of the previous generation. Markov chains describe the probabilistic movement of a particular population (state) to others using a probabilistic transition matrix. In evolutionary algorithms, the transition matrix must express the effects of any selection and variation operators. The transition matrix, when multiplied by itself k times, indicates which new populations can be reached in k generations. This, in principle, allows a calculation of the probability that a population with a solution can be reached. To date a Markov chain for a simplified GP crossover operator has been derived, see [43.142]. Another interesting Markov chain-based

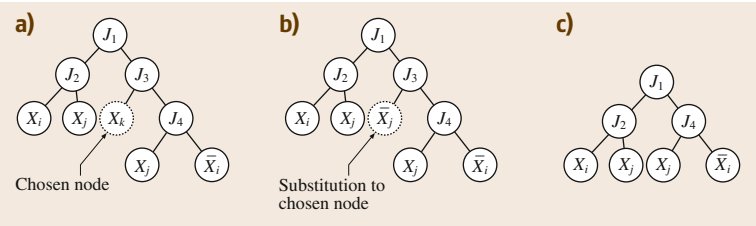

Fig. 43.11a–c HVL-prime mutation: substitution and deletion **(a)** Original parse tree, **(b)** Result of substitution **(c)** Result of deletion

result has revealed that the *distribution of functionality of non-Turing complete programs approaches a limit as length increases*. Markov chain analysis has also been the means of describing what happens with GP semantics rather than syntax. The influence of subtree crossover is studied in a semantic building block analysis by [43.143]. Markov chains, unfortunately, combinatorially explode with even simple extensions of algorithm dynamics or, in GP's case, its theoretically infinite search space. Thus, while they can support further analysis, ultimately this complexity is unwieldy to work with.

Runtime Complexity

Due to stochasticity, it is arguably impossible in most cases to make formal guarantees about the number of fitness evaluations needed for a GP algorithm to find an optimal solution. However, initial steps in the runtime complexity analysis of genetic programming have been made in [43.144]. The authors study the runtime of hill climbing GP algorithms which use a mutation operator called HVL-Prime (Figs. 43.11 and 43.12). Several of these simplified GP algorithms were analyzed on two separable model problems, Order and Majority introduced in [43.145]. Order and Majority each have an independent, additive fitness structure. They each admit multiple solutions based on their objective function, so they exhibit a key property of all real GP problems. They each capture a different relevant facet of typical GP problems. Order represents problems, such as classification problems, where the operators include conditional functions such as an IF-THEN-ELSE. These functions give rise to conditional execution paths which have implications for evolvability and the effectiveness of crossover. Majority is a GP equivalent of the GA OneMax problem [43.146]. It reflects a general (and thus weak) property required of GP solutions: a solution must have correct functionality (by evolving an aggregation of subsolutions) and no incorrect functionality. The analyses highlighted, in particular, the impact of accepting or rejecting neutral moves and the importance of a local mutation operator. A similar finding, [43.147], regarding mutation arose from the analysis of the Max problem [43.148] and hillclimbing. For a search process bounded by a maximally sized tree of n nodes, the time complexity of the simple GP mutation-based hillclimbing algorithms using HVL-Prime for the entire range of MAX variants are $O(n \log^2 n)$ when one mutation operation precedes each fitness evaluation. When multiple mutations are successively applied before each fitness evaluation, the time complexity is $O(n^2)$. This complexity can be reduced to $O(n \log n)$ if the mutations are biased to replace a random leaf with distance d from the root with probability 2^{-d}.

Runtime analyses have also considered parsimony pressure and multiobjective GP algorithms for generalizations of Order and Majority [43.149].

GP algorithms have also been studied in the PAC learning framework [43.150].

Problem Difficulty

Problem difficulty is the study of the differences between algorithms and problems which lead to differences in performance. Stated simply, the goal is to understand why some problems are easy and some are hard, and why some algorithms perform well on certain problems and others do not. Problem difficulty work in the field of GP has much in common with similar work in the broader field of EC. Problem difficulty is naturally related to the size of the search space; smaller spaces are easier to search, as are spaces in which

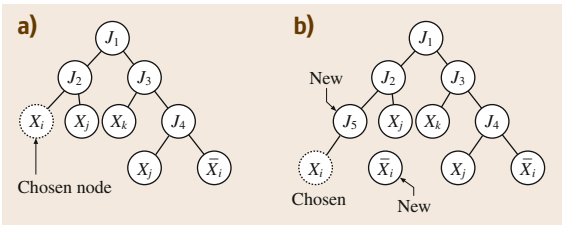

Fig. 43.12a,b HVL-prime mutation: insertion **(a)** Original parse tree, **(b)** Result of insertion

the solution is *over-represented* [43.151]. Difficulty is also related to the *fitness landscape* [43.152], which in turn depends on both the problem and the algorithm and representation chosen to solve it. Landscapes with few local optima (visualized in the fitness landscape as peaks which are not as high as that of the global optimum) are easier to search. *Locality*, that is the property that small changes to a program lead to small changes in fitness, implies a smooth, easily searchable landscape [43.151, 153].

However, more precise statements concerning problem difficulty are usually desired. One important line of research was carried out by *Vanneschi* et al. [43.154–156]. This involved calculating various measures of the *correlation* of the fitness landscape, that is the relationship between distance in the landscape and fitness difference. The measures include the *fitness distance correlation* and the *negative slope coefficient*. These measures require the definition of a distance measure on the search space, which in the case of standard GP means a distance between pairs of trees. Various tree distance measures have been proposed and used for this purpose [43.157–160]. However, the reliable prediction of performance based purely on landscape analysis remains a distant goal in GP as it does in the broader field of EC.

No Free Lunch

In a nutshell, the No Free Lunch Theorem [43.161] proves that, averaged over all problem instances, no algorithm outperforms another. Follow-up NFL analysis [43.162, 163] yields a similar result for problems where the set of fitness functions are closed under permutation. One question is whether the NFL theorem applies to GP algorithms: for some problem class, is it worth developing a better GP algorithm, or will this effort offer no extra value when all instances of the problem are considered? Research has revealed two conditions under which the NFL breaks down for GP because the set of fitness functions is not closed under permutation. First, GP has a many-to-one syntax tree to program output mapping because many different programs have the same functionality while program output functionality is not uniformly distributed across syntax trees. Second, a geometric argument has shown [43.164], that many *realistic* situations exist where a set of GP problems is provably not closed under permutation. The implication of a contradiction to the No Free Lunch theorem is that it is worthwhile investing effort in improving a GP algorithm for a class of problems.

43.5.3 Modularity

Modularity in GP is the ability of a representation to evolve good building blocks and then encapsulate and reuse them. This can be expected to make complex programs far easier to find, since good building blocks needed in multiple places in the program not be laboriously re-evolved each time. One of the best-known approaches to modularity is *automatically defined functions* (ADFs), where the building blocks are implemented as functions which are defined in one part of the evolving program and then invoked from another part [43.58]. This work was followed by automatically defined macros which are more powerful than ADFs and allow control of program flow [43.165]; automatically defined iteration, recursion, and memory stores [43.10]; modularity in other representations [43.166]; and demonstrations of the power of reuse, [43.167].

43.5.4 Open-Ended Evolution and GP

Biological evolution is a long-running exploration of the enormously varied and indefinitely sized DNA search space. There is no hint that a limit on new areas of the space to be explored will ever be reached. In contrast, EC algorithms often operate in search spaces which are finite and highly simplified in comparison to biology. Although GP itself can be used for a wide variety of tasks (Sect. 43.1), each specific instance of the GP algorithm is capable of solving only a very narrow problem. In contrast, some researchers see biological evolution as pointing the way to a more ambitious vision of the possibilities for GP [43.168]. In this vision, an evolutionary run would continue for an indefinite length of time, always exploring new areas of an indefinitely sized search space; always responding to changes in the environment; and always reshaping the search space itself. This vision is particularly well suited to GP, as opposed to GAs and similar algorithms, because GP already works in search spaces which are infinite in theory, if not in practice.

To make this type of GP possible, it is necessary to prevent convergence of the population on a narrow area of the search space. Diversity preservation [43.169], periodic injection of new random material [43.170], and island-structured population models [43.88] can help in this regard.

Open-ended evolution would also be facilitated by complexity and nonstationarity in the algorithm's evo-

lutionary *ecosystem*. If fitness criteria are dynamic or coevolutionary [43.171–173], there may be no natural end-point to evolution, and so continued exploration under different criteria can lead to unlimited new results.

43.6 Practicalities

43.6.1 Conferences and Journals

Several conferences provide venues for the publication of new GP research results. The ACM *Genetic and Evolutionary Computation Conference* (GECCO) alternates annually between North America and the rest of the world and includes a GP track. *EuroGP* is held annually in Europe as the main event of *Evo**, and focuses only on GP. The IEEE *Congress on Evolutionary Computation* is a larger event with broad coverage of EC in general. *Genetic Programming Theory and Practice* is held annually in Ann Arbor, MI, USA and provides a focused forum for GP discussion. *Parallel Problem Solving from Nature* is one of the older, general EC conferences, held biennially in Europe. It alternates with the *Evolution Artificielle* conference. Finally, *Foundations of Genetic Algorithms* is a smaller, theory-focused conference.

The journal most specialized to the field is probably *Genetic Programming and Evolvable Machines* (published by Springer). The September 2010, 10-year anniversary issue included several review articles on GP. *Evolutionary Computation* (MIT Press) and the IEEE *Transactions on Evolutionary Computation* also publish important GP material. Other on-topic journals with a broader focus include *Applied Soft Computing* and *Natural Computing*.

43.6.2 Software

A great variety of GP software is available. We will mention only a few packages – further options can be found online.

One of the well-known Java systems is *ECJ* [43.174, 175]. It is a general-purpose system with support for many representations, problems, and methods, both within GP and in the wider field of EC. It has a helpful mailing list. Watchmaker [43.176] is another general-purpose system with excellent out-of-the-box examples. *GEVA* [43.177, 178] is another Java-based package, this time with support only for GE.

For users of C++ there are also several options. Some popular packages include *Evolutionary Objects* [43.179], μGP [43.180–182], and *OpenBeagle* [43.183, 184]. Matlab users may be interested in GPLab [43.185], which implements standard GP, while DEAP [43.186] provides implementations of several algorithms in Python. PushGP [43.187] is available in many languages.

Two more systems are worth mentioning for their deliberate focus on simplicity and understandability. *TinyGP* [43.188] and *PonyGE* [43.189] implement standard GP and GE respectively, each in a single, readable source file.

Moving on from open source, Michael Schmidt and Hod Lipson's Eureqa [43.190] is a free-to-use tool with a focus on symbolic regression of numerical data and the built-in ability to use cloud resources.

Finally, the authors are aware of two commercially available GP tools, each fast and industrial-strength. They have more automation and *it just works* functionality, relative to most free and open-source tools. Free trials are available. *DataModeler* (Evolved Analytics LLC) [43.191] is a notebook in Mathematica. It employs the ParetoGP method [43.128] which gives the ability to trade program fitness off against complexity, and to form ensembles of programs. It also exploits complex population archiving and archive-based selection. It offers means of dealing with ill-conditioned data and extracting information on variable importance from evolved models. *Discipulus* (Register Machine Learning Technologies, Inc.) [43.192] evolves machine code based on the ideas of *Nordin* et al. [43.193]. It runs on Windows only. The machine code representation allows very fast fitness evaluation and low memory usage, hence large populations. In addition to typical GP features, it can: use an ES to optimise numerical constants; automatically construct ensembles; preprocess data; extract variable importance after runs; automatically simplify results; and save them to high-level languages.

43.6.3 Resources and Further Reading

Another useful resource for GP research is the *GP Bibliography* [43.194]. In addition to its huge, regularly updated collection of BibTeX-formatted citations,

it has lists of researchers' homepages [43.195] and coauthorship graphs. The *GP mailing list* [43.196] is one well-known forum for discussion.

Many of the traditional GP benchmark problems have been criticized for being unrealistic in various ways. The lack of standardization of benchmark problems also allows the possibility of cherry-picking of benchmarks. Effort is underway to bring some standardization to the choice of GP benchmarks [43.197, 198].

Those wishing to read further have many good options. The *Field Guide to GP* is a good introduction, walking the reader through simple examples, scanning large amounts of the literature, and offering practical advice [43.199]. *Luke's Essentials of Metaheuristics* [43.200] also has an introductory style, but is broader in scope. Both are free to download. Other broad and introductory books include those by *Fogel* [43.51] and *Banzhaf* et al. [43.201]. More specialized books include those by *Langdon* and *Poli* [43.202] (coverage of theoretical topics), *Langdon* [43.11] (narrower coverage of GP with data structures), *O'Neill* and *Ryan* [43.77] (GE), *Iba* et al. [43.203] (GP-style ML), and *Sipper* [43.204] (games). *Advances in Genetic Programming*, a series of four volumes, contains important foundational work from the 1990s.

References

43.1 J. McCarthy: Programs with Common Sense, Technical Report (Stanford University, Department of Computer Science, Stanford 1963)

43.2 F. Rosenblatt: The perceptron: A probabilistic model for information storage and organization in the brain, Psychol. Rev. **65**(6), 386 (1958)

43.3 D.E. Rumelhart, J.L. McClelland: *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations* (MIT, Cambridge 1986)

43.4 R.A. Brooks: Intelligence without representation, Artif. Intell. **47**(1), 139–159 (1991)

43.5 C. Cortes, V. Vapnik: Support-vector networks, Mach. Learn. **20**(3), 273–297 (1995)

43.6 L. Page, S. Brin, R. Motwani, T. Winograd: *The Pagerank Citation Ranking: Bringing Order to the Web*, Technical Report 1999-66 (Stanford InfoLab, Stanford 1999), available online at http://ilpubs.stanford.edu:8090/422/. Previous number = SIDL-WP-1999-0120.

43.7 J. Levinson, J. Askeland, J. Becker, J. Dolson, D. Held, S. Kammel, J.Z. Kolter, D. Langer, O. Pink, V. Pratt, M. Sokolsky, G. Stanek, D. Stavens, A. Teichma, M. Werling, S. Thrun: Towards fully autonomous driving: Systems and algorithms, Intell. Veh. Symp. (IV) IEEE (2011) pp. 163–168

43.8 C. Darwin: *The Origin of Species by Means of Natural Selection: Or, the Preservation of Favored Races in the Struggle for Life* (John Murray, London 1859)

43.9 T. Bäck, D.B. Fogel, Z. Michalewicz (Eds.): *Handbook of Evolutionary Computation* (IOP Publ., Bristol 1997)

43.10 J.R. Koza, D. Andre, F.H. Bennett III, M. Keane: *Genetic Programming 3: Darwinian Invention and Problem Solving* (Morgan Kaufman, San Francisco 1999), available online at http://www.genetic-programming.org/gpbook3toc.html

43.11 W.B. Langdon: *Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming!* Genetic Programming, Vol. 1 (Kluwer, Boston 1998), available online at http://www.cs.ucl.ac.uk/staff/W.Langdon/gpdata

43.12 M. Suchorzewski, J. Clune: A novel generative encoding for evolving modular, regular and scalable networks, Proc. 13th Annu. Conf. Genet. Evol. Comput. (2011) pp. 1523–1530

43.13 J. Woodward: Evolving Turing complete representations, Proc. 2003 Congr. Evol. Comput. CEC2003, ed. by R. Sarker, R. Reynolds, H. Abbass, K.C. Tan, B. McKay, D. Essam, T. Gedeon (IEEE, Canberra 2003) pp. 830–837, available online at http://www.cs.bham.ac.uk/~jrw/publications/2003/EvolvingTuringCompleteRepresentations/cec032e.pdf

43.14 J. Tanomaru: Evolving Turing machines from examples, Lect. Notes Comput. Sci. **1363**, 167–180 (1993)

43.15 D. Andre, F.H. Bennett III, J.R. Koza: Discovery by genetic programming of a cellular automata rule that is better than any known rule for the majority classification problem, Proc. 1st Annu. Conf. Genet. Progr., ed. by J.R. Koza, D.E. Goldberg, D.B. Fogel, R.L. Riolo (MIT Press, Cambridge 1996) pp. 3–11, available online at http://www.genetic-programming.com/jkpdf/gp1996gkl.pdf

43.16 F. Gruau: Neural Network Synthesis Using Cellular Encoding and the Genetic Algorithm, Ph.D. Thesis (Laboratoire de l'Informatique du Parallilisme, Ecole Normale Supirieure de Lyon, France 1994), available online at ftp://ftp.ens-lyon.fr/pub/LIP/Rapports/PhD/PhD1994/PhD1994-01-E.ps.Z

43.17 A. Teller: Turing completeness in the language of genetic programming with indexed memory, Proc. 1994 IEEE World Congr. Comput. Intell., Orlando, Vol. 1 (1994) pp. 136–141, available online at http://www.cs.cmu.edu/afs/cs/usr/astro/public/papers/Turing.ps

43.18 S. Mabu, K. Hirasawa, J. Hu: A graph-based evolutionary algorithm: Genetic network programming (GNP) and its extension using reinforcement learning, Evol. Comput. **15**(3), 369–398 (2007)

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

43.20 P. Prusinkiewicz, A. Lindenmayer: *The Algorithmic Beauty of Plants (The Virtual Laboratory)* (Springer, Berlin, Heidelberg 1991)

43.21 J. Murphy, M. O'Neill, H. Carr: Exploring grammatical evolution for horse gait optimisation, Lect. Notes Comput. Sci. **5481**, 183–194 (2009)

43.22 T. Haynes, S. Sen: Evolving behavioral strategies in predators and prey, Lect. Notes Comput. Sci. **1042**, 113–126 (1995)

43.23 R. De Caux: Using Genetic Programming to Evolve Strategies for the Iterated Prisoner's Dilemma, Master's Thesis (University College, London 2001), available online at http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/decaux.masters.zip

43.24 A. Hauptman, M. Sipper: GP-endchess: Using genetic programming to evolve chess endgame players, Lect. Notes Comput. Sci. **3447**, 120–131 (2005), available online at http://www.cs.bgu.ac.il/~sipper/papabs/eurogpchess-final.pdf

43.25 E. Galván-Lopéz, J.M. Swafford, M. O'Neill, A. Brabazon: Evolving a Ms. PacMan controller using grammatical evolution, Lect. Notes Comput. Sci. **6024**, 161–170 (2010)

43.26 J. Togelius, S. Lucas, H.D. Thang, J.M. Garibaldi, T. Nakashima, C.H. Tan, I. Elhanany, S. Berant, P. Hingston, R.M. MacCallum, T. Haferlach, A. Gowrisankar, P. Burrow: The 2007 IEEE CEC simulated car racing competition, Genet. Program. Evol. Mach. **9**(4), 295–329 (2008)

43.27 G.S. Hornby, J.D. Lohn, D.S. Linden: Computer-automated evolution of an X-band antenna for NASA's space technology 5 mission, Evol. Comput. **19**(1), 1–23 (2011)

43.28 M. Furuholmen, K.H. Glette, M.E. Hovin, J. Torresen: Scalability, generalization and coevolution – experimental comparisons applied to automated facility layout planning, GECCO '09: Proc. 11th Annu. Conf. Genet. Evol. Comput., Montreal, ed. by F. Rothlauf, G. Raidl (2009) pp. 691–698, available online at http://doi.acm.org/10.1145/1569901.1569997

43.29 C.C. Bojarczuk, H.S. Lopes, A.A. Freitas: Genetic programming for knowledge discovery in chest-pain diagnosis, IEEE Eng. Med. Biol. Mag. **19**(4), 38–44 (2000), available online at http://ieeexplore.ieee.org/iel5/51/18543/00853480.pdf

43.30 T. Hildebrandt, J. Heger, B. Scholz-Reiter, M. Pelikan, J. Branke: Towards improved dispatching rules for complex shop floor scenarios: A genetic programming approach, GECCO '10: Proc. 12th Annu. Conf. Genet. Evol. Comput., Portland, ed. by J. Branke (2010) pp. 257–264

43.31 M.B. Bader-El-Den, R. Poli, S. Fatima: Evolving timetabling heuristics using a grammar-based genetic programming hyper-heuristic framework, Memet. Comput. **1**(3), 205–219 (2009), 10.1007/s12293-009-0022-y

43.32 M. Conrads, P. Nordin, W. Banzhaf: Speech sound discrimination with genetic programming, Lect. Notes Comput. Sci. **1391**, 113–129 (1998)

43.33 A. Esparcia-Alcazar, K. Sharman: Genetic programming for channel equalisation, Lect. Notes Comput. Sci. **1596**, 126–137 (1999), available online at http://www.iti.upv.es/~anna/papers/evoiasp99.ps

43.34 R. Poli, M. Salvaris, C. Cinel: Evolution of a brain-computer interface mouse via genetic programming, Lect. Notes Comput. Sci. **6621**, 203–214 (2011)

43.35 K. Sims: Artificial evolution for computer graphics, ACM Comput. Gr. **25**(4), 319–328 (1991), available online at http://delivery.acm.org/10.1145/130000/122752/p319-sims.pdf SIGGRAPH '91 Proceedings

43.36 U.-M. O'Reilly, M. Hemberg: Integrating generative growth and evolutionary computation for form exploration, Genet. Program. Evol. Mach. **8**(2), 163–186 (2007), Special issue on developmental systems

43.37 P. Dahlstedt: Autonomous evolution of complete piano pieces and performances, Proc. Music AL Workshop (2007)

43.38 H. Iba: Multiple-agent learning for a robot navigation task by genetic programming, Genet. Program. Proc. 2nd Annu. Conf., Standord, ed. by J.R. Koza, K. Deb, M. Dorigo, D.B. Fogel, M. Garzon, H. Iba, R.L. Riolo (1997) pp. 195–200

43.39 T. Weise, K. Tang: Evolving distributed algorithms with genetic programming, IEEE Trans. Evol. Comput. **16**(2), 242–265 (2012)

43.40 L. Spector: Autoconstructive evolution: Push, pushGP, and pushpop, Proc. Genet. Evol. Comput. Conf. (GECCO-2001), ed. by L. Spector, E. Goodman (Morgan Kaufmann, San Francisco 2001) pp. 137–146, available online at http://hampshire.edu/lspector/pubs/ace.pdf

43.41 J. Tavares, F. Pereira: Automatic design of ant algorithms with grammatical evolution. In: *Gnetic Programming. 15th European Conference, EuroGP*, ed. by A. Moraglio, S. Silva, K. Krawiec, P. Machado, C. Cotta (Springer, Berlin, Heidelberg 2012) pp. 206–217

43.42 M. Hutter: *A Gentle Introduction To The Universal Algorithmic Agent* AIXI. Technical Report IDSIA-01-03 (IDSIA, Manno-Lugano 2003)

43.43 J. Von Neumann, M.D. Godfrey: First draft of a report on the EDVAC, IEEE Ann. Hist. Comput. **15**(4), 27–75 (1993)

43.44 H.H. Goldstine, A. Goldstine: The electronic numerical integrator and computer (ENIAC), Math. Tables Other Aids Comput. **2**(15), 97–110 (1946)

43.45 A.M. Turing: Intelligent machinery. In: *Cybernetics: Key Papers*, ed. by C.R. Evans, A.D.J. Robert-

son (Univ. Park Press, Baltimore 1968), Written 1948
43.46 A.M. Turing: Computing machinery and intelligence, Mind **59**(236), 433–460 (1950)
43.47 A.L. Samuel: Some studies in machine learning using the game of checkers, IBM J. Res. Dev. **3**(3), 210 (1959)
43.48 R.M. Friedberg: A learning machine: Part I, IBM J. Res. Dev. **2**(1), 2–13 (1958)
43.49 M. O'Neill, L. Vanneschi, S. Gustafson, W. Banzhaf: Open issues in genetic programming, Genet. Program. Evol. Mach. **11**(3/4), 339–363 (2010), 10th Anniversary Issue: Progress in Genetic Programming and Evolvable Machines
43.50 L.J. Fogel, A.J. Owens, M.J. Walsh: *Artificial Intelligence Through Simulated Evolution* (Wiley, Hoboken 1966)
43.51 D.B. Fogel: *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*, Vol. 1 (Wiley, Hoboken 2006)
43.52 S.F. Smith: A Learning System Based on Genetic Adaptive Algorithms, Ph.D. Thesis (University of Pittsburgh, Pittsburgh 1980)
43.53 N.L. Cramer: A representation for the adaptive generation of simple sequential programs, Proc. Int. Conf. Genet. Algorithms Appl., Pittsburgh, ed. by J.J. Grefenstette (1985) pp. 183–187, available online at http://www.sover.net/~nichael/nlc-publications/icga85/index.html
43.54 J. Schmidhuber: Evolutionary Principles in Self-Referential Learning. On Learning Now to Learn: The Meta-Meta-Meta...-Hook, Diploma Thesis (Technische Universität, München 1987), available online at http://www.idsia.ch/~juergen/diploma.html
43.55 C. Fujiki, J. Dickinson: Using the genetic algorithm to generate lisp source code to solve the prisoner's dilemma, Proc. 2nd Int. Conf. Genet. Algorithms Appl., Cambridge, ed. by J.J. Grefenstette (1987) pp. 236–240
43.56 A.S. Bickel, R.W. Bickel: Tree structured rules in genetic algorithms, Proc. 2nd Int. Conf. Genet. Algorithms Appl., Cambridge, ed. by J.J. Grefenstette (1987) pp. 77–81
43.57 T.S. Ray: Evolution, Ecology and Optimization of Digital Organisms. Technical Report Working Paper 92-08-042 (Santa Fe Institute, Santa Fe 1992) available online at http://www.santafe.edu/media/workingpapers/92-08-042.pdf
43.58 J.R. Koza: *Genetic Programming II: Automatic Discovery of Reusable Programs* (MIT, Cambridge 1994)
43.59 J.R. Koza, M.A. Keane, M.J. Streeter, W. Mydlowec, J. Yu, G. Lanza: *Genetic Programming IV: Routine Human-Competitive Machine Intelligence* (Springer, Berlin, Heidelberg 2003), available online at http://www.genetic-programming.org/gpbook4toc.html
43.60 J. Koza: http://www.genetic-programming.org/hc2011/combined.html
43.61 J.G. Carbonell, R.S. Michalski, T.M. Mitchell: An overview of machine learning. In: *Machine Learning: An Artificial Intelligence Approach*, ed. by R.S. Michalski, J.G. Carbonell, T.M. Mitchell (Tioga, Palo Alto 1983)
43.62 C. Rich, R.C. Waters: Automatic programming: Myths and prospects, Computer **21**(8), 40–51 (1988)
43.63 S. Gulwani: Dimensions in program synthesis, Proc. 12th Int. SIGPLAN Symp. Princ. Pract. Declar. Program. (2010) pp. 13–24
43.64 I. Rechenberg: *Evolutionsstrategie – Optimierung Technischer Systeme nach Prinzipien der Biologischen Evolution* (Frommann-Holzboog, Stuttgart 1973)
43.65 H.-P. Schwefel: *Numerische Optimierung von Computer-Modellen* (Birkhäuser, Basel 1977)
43.66 J.H. Holland: *Adaptation in Natural and Artificial Systems* (University of Michigan, Ann Arbor 1975)
43.67 D.J. Montana: Strongly typed genetic programming, Evol. Comput. **3**(2), 199–230 (1995), available online at http://vishnu.bbn.com/papers/stgp.pdf
43.68 T. Yu: Hierachical processing for evolving recursive and modular programs using higher order functions and lambda abstractions, Genet. Program. Evol. Mach. **2**(4), 345–380 (2001)
43.69 R. Poli: Parallel distributed genetic programming. In: *New Ideas in Optimization*, Advanced Topics in Computer Science, ed. by D. Corne, M. Dorigo, F. Glover (McGraw-Hill, London 1999) pp. 403–431, Chapter 27, available online at http://citeseer.ist.psu.edu/328504.html
43.70 J.F. Miller, P. Thomson: Cartesian genetic programming, Lect. Notes Comput. Sci. **1802**, 121–132 (2000), available online at http://www.elec.york.ac.uk/intsys/users/jfm7/cgp-eurogp2000.pdf
43.71 K.O. Stanley: Compositional pattern producing networks: A novel abstraction of development, Genet. Program. Evol. Mach. **8**(2), 131–162 (2007)
43.72 L.J. Fogel, P.J. Angeline, D.B. Fogel: An evolutionary programming approach to self-adaptation on finite state machines, Proc. 4th Int. Conf. Evol. Program. (1995) pp. 355–365
43.73 R.I. McKay, N.X. Hoai, P.A. Whigham, Y. Shan, M. O'Neill: Grammar-based genetic programming: A survey, Genet. Program. Evol. Mach. **11**(3/4), 365–396 (2010), September Tenth Anniversary Issue: Progress in Genetic Programming and Evolvable Machines
43.74 M. Keijzer, V. Babovic: Dimensionally aware genetic programming, Proc. Genet. Evol. Comput. Conf., Orlando, Vol. 2, ed. by W. Banzhaf, J. Daida, A.E. Eiben, M.H. Garzon, V. Honavar, M. Jakiela, R.E. Smith (1999) pp. 1069–1076, available online at http://www.cs.bham.ac.uk/~wbl/biblio/gecco1999/GP-420.ps

43.75 A. Ratle, M. Sebag: Grammar-guided genetic programming and dimensional consistency: Application to non-parametric identification in mechanics, Appl. Soft Comput. **1**(1), 105–118 (2001), available online at http://www.sciencedirect.com/science/article/B6W86-43S6W98-B/1/38e0fa6ac503a5ef310e2287be01eff8

43.76 P.A. Whigham: Grammatically-based genetic programming, Proc. Workshop Genet. Program.: From Theory Real-World Appl., Tahoe City, ed. by J.P. Rosca (1995) pp. 33–41, available online at http://divcom.otago.ac.nz/sirc/Peterw/Publications/ml95.zip

43.77 M. O'Neill, C. Ryan: *Grammatical Evolution: Evolutionary Automatic Programming in a Arbitrary Language*, Genetic Programming, Vol. 4 (Kluwer, Boston 2003), available online at http://www.wkap.nl/prod/b/1-4020-7444-1

43.78 N. Xuan Hoai, R.I. McKay, D. Essam: Representation and structural difficulty in genetic programming, IEEE Trans. Evol. Comput. **10**(2), 157–166 (2006), available online at http://sc.snu.ac.kr/courses/2006/fall/pg/aai/GP/nguyen/Structdiff.pdf

43.79 Y. Shan, R.I. McKay, D. Essam, H.A. Abbass: A survey of probabilistic model building genetic programming. In: *Scalable Optimization via Probabilistic Modeling: From Algorithms to Applications*, Studies in Computational Intelligence, Vol. 33, ed. by M. Pelikan, K. Sastry, E. Cantu-Paz (Springer, Berlin, Heidelberg 2006) pp. 121–160, Chapter 6

43.80 M. Brameier, W. Banzhaf: *Linear Genetic Programming*, Genetic and Evolutionary Computation, Vol. 16 (Springer, Berlin, Heidelberg 2007), available online at http://www.springer.com/west/home/default?SGWID=4-40356-22-173660820-0

43.81 T. Perkis: Stack-based genetic programming, Proc. 1994 IEEE World Congr. Comput. Intell., Orlando, Vol. 1 (1994) pp. 148–153, available online at http://citeseer.ist.psu.edu/432690.html

43.82 W.B. Langdon: Large scale bioinformatics data mining with parallel genetic programming on graphics processing units. In: *Parallel and Distributed Computational Intelligence*, Studies in Computational Intelligence, Vol. 269, ed. by F. de Fernandez Vega, E. Cantu-Paz (Springer, Berlin, Heidelberg 2010) pp. 113–141, Chapter 5, available online at http://www.springer.com/engineering/book/978-3-642-10674-3

43.83 L. Spector, A. Robinson: Genetic programming and autoconstructive evolution with the push programming language, Genet. Program. Evol. Mach. **3**(1), 7–40 (2002), available online at http://hampshire.edu/lspector/pubs/push-gpem-final.pdf

43.84 E. Schulte, S. Forrest, W. Weimer: Automated program repair through the evolution of assembly code, Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng. (2010) pp. 313–316

43.85 M. Orlov, M. Sipper: Flight of the FINCH through the Java wilderness, IEEE Trans. Evol. Comput. **15**(2), 166–182 (2011)

43.86 P. Nordin: A compiling genetic programming system that directly manipulates the machine code. In: *Advances in Genetic Programming*, ed. by K.E. Kinnear Jr. (MIT Press, Cambridge 1994) pp. 311–331, Chapter 14, available online at http://cognet.mit.edu/library/books/view?isbn=0262111888

43.87 U.-M. O'Reilly, F. Oppacher: Program search with a hierarchical variable length representation: Genetic programming, simulated annealing and hill climbing, Lect. Notes Comput. Sci. **866**, 397–406 (1994), available online at http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/ppsn-94.ps.gz

43.88 M. Tomassini: *Spatially Structured Evolutionary Algorithms* (Springer, Berlin, Heidelberg 2005)

43.89 A. Arcuri, X. Yao: A novel co-evolutionary approach to automatic software bug fixing, IEEE World Congr. Comput. Intell., Hong Kong, ed. by J. Wang (2008)

43.90 A. Moraglio, C. Di Chio, R. Poli: Geometric particle swarm optimization, Lect. Notes Comput. Sci. **4445**, 125–136 (2007)

43.91 M. O'Neill, A. Brabazon: Grammatical differential evolution, Proc. Int. Conf. Artif. Intell. ICAI 2006, Las Vegas, Vol. 1, ed. by H.R. Arabnia (2006) pp. 231–236, available online at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.91.3012

43.92 R. Poli, N.F. McPhee: A linear estimation-of-distribution GP system, Lect. Notes Comput. Sci. **4971**, 206–217 (2008)

43.93 M. Looks, B. Goertzel, C. Pennachin: Learning computer programs with the Bayesian optimization algorithm, GECCO 2005: Proc. Conf. Genet. Evol. Comput., Washington, Vol. 1, ed. by U.-M. O'Reilly, H.-G. Beyer (2005) pp. 747–748, available online at http://www.cs.bham.ac.uk/~wbl/biblio/gecco2005/docs/p747.pdf

43.94 E. Hemberg, K. Veeramachaneni, J. McDermott, C. Berzan, U.-M. O'Reilly: An investigation of local patterns for estimation of distribution genetic programming, Philadelphia, Proc. GECCO 2012 (2012)

43.95 M. Schmidt, H. Lipson: Distilling free-form natural laws from experimental data, Science **324**(5923), 81–85 (2009), available online at http://ccsl.mae.cornell.edu/sites/default/files/Science09_Schmidt.pdf

43.96 E.J. Vladislavleva, G.F. Smits, D. den Hertog: Order of nonlinearity as a complexity measure for models generated by symbolic regression via Pareto genetic programming, IEEE Trans. Evol. Comput. **13**(2), 333–349 (2009)

43.97 K. Veeramachaneni, E. Vladislavleva, U.-M. O'Reilly: Knowledge mining sensory

evaluation data: Genetic programming, statistical techniques, and swarm optimization, Genet. Program. Evolvable Mach. **13**(1), 103–133 (2012)

43.98 M. Kotanchek, G. Smits, E. Vladislavleva: Pursuing the Pareto paradigm tournaments, algorithm variations & ordinal optimization. In: *Genetic Programming Theory and Practice IV*, Genetic and Evolutionary Computation, Vol. 5, ed. by R.L. Riolo, T. Soule, B. Worzel (Springer, Berlin, Heidelberg 2006) pp. 167–186, Chapter 12

43.99 C. Le Goues, T. Nguyen, S. Forrest, W. Weimer: GenProg: A generic method for automated software repair, IEEE Trans. Softw. Eng. **38**(1), 54–72 (2011)

43.100 C. Ryan: *Automatic Re-Engineering of Software Using Genetic Programming*, Genetic Programming, Vol. 2 (Kluwer, Boston 2000), available online at http://www.wkap.nl/book.htm/0-7923-8653-1

43.101 K.P. Williams: Evolutionary Algorithms for Automatic Parallelization, Ph.D. Thesis (University of Reading, Reading 1998)

43.102 M. Stephenson, S. Amarasinghe, M. Martin, U.-M. O'Reilly: Meta optimization: Improving compiler heuristics with machine learning, Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implement. (PLDI '03), San Diego (2003) pp. 77–90

43.103 M. Stephenson, U.-M. O'Reilly, M.C. Martin, S. Amarasinghe: Genetic programming applied to compiler heuristic optimization, Lect. Notes Comput. Sci. **2610**, 238–253 (2003)

43.104 D.R. White, A. Arcuri, J.A. Clark: Evolutionary improvement of programs, IEEE Trans. Evol. Comput. **15**(4), 515–538 (2011)

43.105 M. Harman: The current state and future of search based software engineering, Proc. Future of Software Engineering FOSE '07, Washington, ed. by L. Briand, A. Wolf (2007) pp. 342–357

43.106 J.R. Koza, F.H. Bennett III, D. Andre, M.A. Keane, F. Dunlap: Automated synthesis of analog electrical circuits by means of genetic programming, IEEE Trans. Evol. Comput. **1**(2), 109–128 (1997), available online at http://www.genetic-programming.com/jkpdf/ieeetecjournal1997.pdf

43.107 R. Dawkins: *The Blind Watchmaker* (Norton, New York 1986)

43.108 H. Takagi: Interactive evolutionary computation: Fusion of the capabilities of EC optimization and human evaluation, Proc. IEEE **89**(9), 1275–1296 (2001)

43.109 S. Todd, W. Latham: *Evolutionary Art and Computers* (Academic, Waltham 1994)

43.110 M. Lewis: Evolutionary visual art and design. In: *The Art of Artificial Evolution: A Handbook on Evolutionary Art and Music*, ed. by J. Romero, P. Machado (Springer, Berlin, Heidelberg 2008) pp. 3–37

43.111 J. McDermott, J. Byrne, J.M. Swafford, M. O'Neill, A. Brabazon: Higher-order functions in aesthetic EC encodings, 2010 IEEE World Congr. Comput. Intell., Barcelona (2010), pp. 2816–2823, 18-23 July

43.112 D.A. Hart: Toward greater artistic control for interactive evolution of images and animation, Lect. Notes Comput. Sci. **4448**, 527–536 (2007)

43.113 A.K. Hoover, M.P. Rosario, K.O. Stanley: Scaffolding for interactively evolving novel drum tracks for existing songs, Lect. Notes Comput. Sci. **4974**, 412 (2008)

43.114 J. Shao, J. McDermott, M. O'Neill, A. Brabazon: Jive: A generative, interactive, virtual, evolutionary music system, Lect. Notes Comput. Sci. **6025**, 341–350 (2010)

43.115 J. McDermott, U.-M. O'Reilly: An executable graph representation for evolutionary generative music, Proc. GECCO 2011 (2011) pp. 403–410

43.116 J. Clune, H. Lipson: Evolving three-dimensional objects with a generative encoding inspired by developmental biology, Proc. Eur. Conf. Artif. Life (2011), available online at http://endlessforms.com

43.117 J. McCormack: Evolutionary L-systems. In: *Design by Evolution: Advances in Evolutionary Design*, ed. by P.F. Hingston, L.C. Barone, Z. Michalewicz, D.B. Fogel (Springer, Berlin, Heidelberg 2008) pp. 169–196

43.118 G.S. Hornby, J.B. Pollack: Evolving L-systems to generate virtual creatures, Comput. Graph. **25**(6), 1041–1048 (2001)

43.119 P. Worth, S. Stepney: Growing music: Musical interpretations of L-systems, Lect. Notes Comput. Sci. **3449**, 545–550 (2005)

43.120 J. McDermott, J. Byrne, J.M. Swafford, M. Hemberg, C. McNally, E. Shotton, E. Hemberg, M. Fenton, M. O'Neill: String-rewriting grammars for evolutionary architectural design, Environ. Plan. B **39**(4), 713–731 (2012), available online at http://www.envplan.com/abstract.cgi?id=b38037

43.121 W. Banzhaf, W.B. Langdon: Some considerations on the reason for bloat, Genet. Program. Evol. Mach. **3**(1), 81–91 (2002), available online at http://web.cs.mun.ca/~banzhaf/papers/genp_bloat.pdf

43.122 S. Luke, L. Panait: A comparison of bloat control methods for genetic programming, Evol. Comput. **14**(3), 309–344 (2006)

43.123 S. Silva, S. Dignum, L. Vanneschi: Operator equalisation for bloat free genetic programming and a survey of bloat control methods, Genet. Program. Evol. Mach. **3**(2), 197–238 (2011)

43.124 W.B. Langdon, R. Poli: Fitness causes bloat. In: *Soft Computing in Engineering Design and Manufacturing*, ed. by P.K. Chawdhry, R. Roy, R.K. Pant (Springer, London 1997) pp. 13–22, available online at http://www.cs.bham.ac.uk/~wbl/ftp/papers/WBL.bloat_wsc2.ps.gz

43.125 S. Dignum, R. Poli: Generalisation of the limiting distribution of program sizes in tree-based genetic programming and analysis of its effects on bloat, GECCO '07 Proc. 9th Annu. Conf. Genet. Evol. Comput., London, Vol. 2, ed. by H. Lipson, D. Thierens (2007) pp. 1588–1595, available online at http://www.cs.bham.ac.uk/~wbl/biblio/gecco2007/docs/p1588.pdf

43.126 W.B. Langdon: How many good programs are there? How long are they?, Found. Genet. Algorithms VII, San Francisco, ed. by K.A. De Jong, R. Poli, J.E. Rowe (2002), pp. 183–202, available online at http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/wbl_foga2002.pdf

43.127 R. Poli, M. Salvaris, C. Cinel: Evolution of an effective brain-computer interface mouse via genetic programming with adaptive Tarpeian bloat control. In: *Genetic Programming Theory and Practice IX*, ed. by R. Riolo, K. Vladislavleva, J. Moore (Springer, Berlin, Heidelberg 2011) pp. 77–95

43.128 G. Smits, E. Vladislavleva: Ordinal pareto genetic programming, Proc. 2006 IEEE Congr. Evol. Comput., Vancouver, ed. by G.G. Yen, S.M. Lucas, G. Fogel, G. Kendall, R. Salomon, B.-T. Zhang, C.A. Coello Coello, T.P. Runarsson (2006) pp. 3114–3120, available online at http://ieeexplore.ieee.org/servlet/opac?punumber=11108

43.129 L. Vanneschi, M. Castelli, S. Silva: Measuring bloat, overfitting and functional complexity in genetic programming, GECCO '10: Proc. 12th Annu. Conf. Genet. Evol. Comput., Portland (2010) pp. 877–884

43.130 H. Iba, H. de Garis, T. Sato: Genetic programming using a minimum description length principle. In: *Advances in Genetic Programming*, ed. by K.E. Kinnear Jr. (MIT Press, Cambridge 1994) pp. 265–284, available online at http://citeseer.ist.psu.edu/327857.html, Chapter 12

43.131 J. Miller: What bloat? Cartesian genetic programming on boolean problems, 2001 Genet. Evol. Comput. Conf. Late Break. Pap., ed. by E.D. Goodman (2001) pp. 295–302, available online at http://www.elec.york.ac.uk/intsys/users/jfm7/gecco2001Late.pdf

43.132 R. Poli, J. Page, W.B. Langdon: Smooth uniform crossover, sub-machine code GP and demes: A recipe for solving high-order Boolean parity problems, Proc. Genet. Evol. Comput. Conf., Orlando, Vol. 2, ed. by W. Banzhaf, J. Daida, A.E. Eiben, M.H. Garzon, V. Honavar, M. Jakiela, R.E. Smith (1999) pp. 1162–1169, available online at http://www.cs.bham.ac.uk/~wbl/biblio/gecco1999/GP-466.pdf

43.133 M. Keijzer: Alternatives in subtree caching for genetic programming, Lect. Notes Comput. Sci. **3003**, 328–337 (2004), available online at http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=3003&spage=328

43.134 R. Poli, W.B. Langdon: Running genetic programming backward. In: *Genetic Programming Theory and Practice III*, Genetic Programming, Vol. 9, ed. by T. Yu, R.L. Riolo, B. Worzel (Springer, Berlin, Heidelberg 2005) pp. 125–140, Chapter 9, available online at http://www.cs.essex.ac.uk/staff/poli/papers/GPTP2005.pdf

43.135 Q.U. Nguyen, X.H. Nguyen, M. O'Neill, R.I. McKay, E. Galván-López: Semantically-based crossover in genetic programming: Application to real-valued symbolic regression, Genet. Program. Evol. Mach. **12**, 91–119 (2011)

43.136 L. Altenberg: Emergent phenomena in genetic programming, Evol. Progr. – Proc. 3rd Annu. Conf., San Diego, ed. by A.V. Sebald, L.J. Fogel (1994) pp. 233–241, available online at http://dynamics.org/~altenber/PAPERS/EPIGP/

43.137 U.-M. O'Reilly, F. Oppacher: The troubling aspects of a building block hypothesis for genetic programming, Working Paper 94-02-001 (Santa Fe Institute, Santa Fe 1992)

43.138 R. Poli, W.B. Langdon: A new schema theory for genetic programming with one-point crossover and point mutation, Proc. Second Annu. Conf. Genet. Progr. 1997, Stanford, ed. by J.R. Koza, K. Deb, M. Dorigo, D.B. Fogel, M. Garzon, H. Iba, R.L. Riolo (1997) pp. 278–285, available online at http://citeseer.ist.psu.edu/327495.html

43.139 J.P. Rosca: Analysis of complexity drift in genetic programming, Proc. 2nd Annu. Conf. Genet. Program. 1997, Stanford, ed. by J.R. Koza, K. Deb, M. Dorigo, D.B. Fogel, M. Garzon, H. Iba, R.L. Riolo (1997), pp. 286–294, available online at ftp://ftp.cs.rochester.edu/pub/u/rosca/gp/97.gp.ps.gz

43.140 R. Poli, N.F. McPhee: General schema theory for genetic programming with subtree-swapping crossover: Part I, Evol. Comput. **11**(1), 53–66 (2003), available online at http://cswww.essex.ac.uk/staff/rpoli/papers/ecj2003partI.pdf

43.141 R. Poli, N.F. McPhee: General schema theory for genetic programming with subtree-swapping crossover: Part II, Evol. Comput. **11**(2), 169–206 (2003), available online at http://cswww.essex.ac.uk/staff/rpoli/papers/ecj2003partII.pdf

43.142 R. Poli, N.F. McPhee, J.E. Rowe: Exact schema theory and Markov chain models for genetic programming and variable-length genetic algorithms with homologous crossover, Genet. Program. Evol. Mach. **5**(1), 31–70 (2004), available online at http://cswww.essex.ac.uk/staff/rpoli/papers/GPEM2004.pdf

43.143 N.F. McPhee, B. Ohs, T. Hutchison: Semantic building blocks in genetic programming, Lect. Notes Comput. Sci. **4971**, 134–145 (2008)

43.144 G. Durrett, F. Neumann, U.-M. O'Reilly: Computational complexity analysis of simple genetic programming on two problems modeling isolated program semantics, Proc. 11th Workshop Found. Genet. Algorithm. (ACM, New York 2011) pp. 69–

80, available online at http://arxiv.org/pdf/1007.4636v1 arXiv:1007.4636v1
43.145 D.E. Goldberg, U.-M. O'Reilly: Where does the good stuff go, and why? How contextual semantics influence program structure in simple genetic programming, Lect. Notes Comput. Sci. **1391**, 16–36 (1998), available online at http://citeseer.ist.psu.edu/96596.html
43.146 D.E. Goldberg: *Genetic Algorithms in Search, Optimization, and Machine Learning* (Addison-Wesley, Reading 1989)
43.147 T. Kötzing, F. Neumann, A. Sutton, U.-M. O'Reilly: The max problem revisited: The importance of mutation in genetic programming, GECCO '12 Proc. 14th Annu. Conf. Genet. Evolut. Comput. (ACM, New York 2012) pp. 1333–1340
43.148 C. Gathercole, P. Ross: *The Max Problem for Genetic Programming – Highlighting an Adverse Interaction Between the Crossover Operator and a Restriction on Tree Depth*, Technical Report (Department of Artificial Intelligence, University of Edinburgh, Edinburgh 1995) available online at http://citeseer.ist.psu.edu/gathercole95max.html
43.149 F. Neumann: Computational complexity analysis of multi-objective genetic programming, GECCO '12 Proc. 14th Annu. Conf. Genet. Evolut. Comput. (ACM, New York 2012) pp. 799–806
43.150 T. Kötzing, F. Neumann, R. Spöhel: PAC learning and genetic programming, Proc. 13th Annu. Conf. Genet. Evol. Comput. (ACM, New York 2011) pp. 2091–2096
43.151 F. Rothlauf: *Representations for Genetic and Evolutionary Algorithms*, 2nd edn. (Physica, Heidelberg 2006)
43.152 T. Jones: Evolutionary Algorithms, Fitness Landscapes and Search, Ph.D. Thesis (University of New Mexico, Albuquerque 1995)
43.153 J. McDermott, E. Galván-Lopéz, M. O'Neill: A fine-grained view of phenotypes and locality in genetic programming. In: *Genetic Programming Theory and Practice*, Vol. 9, ed. by R. Riolo, K. Vladislavleva, J. Moore (Springer, Berlin, Heidelberg 2011)
43.154 M. Tomassini, L. Vanneschi, P. Collard, M. Clergue: A study of fitness distance correlation as a difficulty measure in genetic programming, Evol. Comput. **13**(2), 213–239 (2005)
43.155 L. Vanneschi: Theory and Practice for Efficient Genetic Programming, Ph.D. Thesis (Université de Lausanne, Lausanne 2004)
43.156 L. Vanneschi, M. Tomassini, P. Collard, S. Verel, Y. Pirola, G. Mauri: A comprehensive view of fitness landscapes with neutrality and fitness clouds, Lect. Notes Comput. Sci. **4445**, 241–250 (2007)
43.157 A. Ekárt, S.Z. Németh: A metric for genetic programs and fitness sharing, Lect. Notes Comput. Sci. **1802**, 259–270 (2000)
43.158 S. Gustafson, L. Vanneschi: Crossover-based tree distance in genetic programming, IEEE Trans. Evol. Comput. **12**(4), 506–524 (2008)
43.159 J. McDermott, U.-M. O'Reilly, L. Vanneschi, K. Veeramachaneni: How far is it from here to there? A distance that is coherent with GP operators, Lect. Notes Comput. Sci. **6621**, 190–202 (2011)
43.160 U.-M. O'Reilly: Using a distance metric on genetic programs to understand genetic operators, Int. Conf. Syst. Man Cybern. Comput. Cybern. Simul. (1997) pp. 233–241
43.161 D.H. Wolpert, W.G. Macready: No free lunch theorems for optimization, Evol. Comput. IEEE Trans. **1**(1), 67–82 (1997)
43.162 C. Schumacher, M.D. Vose, L.D. Whitley: The no free lunch and problem description length, Proc. Genet. Evol. Comput. Conf. GECCO-2001 (2001) pp. 565–570
43.163 J.R. Woodward, J.R. Neil: No free lunch, program induction and combinatorial problems, Lect. Notes Comput. Sci. **2610**, 475–484 (2003)
43.164 R. Poli, M. Graff, N.F. McPhee: Free lunches for function and program induction, FOGA '09: Proc. 10th ACM SIGEVO Workshop Found. Genet. Algorithms, Orlando (2009) pp. 183–194
43.165 L. Spector: Simultaneous evolution of programs and their control structures. In: *Advances in Genetic Programming*, Vol. 2, ed. by P.J. Angeline, K.E. Kinnear Jr. (MIT, Cambridge 1996) pp. 137–154, Chapter 7, available online at http://helios.hampshire.edu/lspector/pubs/AiGP2-post-final-e.pdf
43.166 L. Spector, B. Martin, K. Harrington, T. Helmuth: Tag-based modules in genetic programming, Proc. Genet. Evol. Comput. Conf. GECCO-2011 (2011)
43.167 G.S. Hornby: Measuring, nabling and comparing modularity, regularity and hierarchy in evolutionary design, GECCO 2005: Proc. 2005 Conf. Genet. Evol. Comput., Washington, Vol. 2, ed. by H.-G. Beyer, U.-M. O'Reilly, D.V. Arnold, W. Banzhaf, C. Blum, E.W. Bonabeau, E. Cantu-Paz, D. Dasgupta, K. Deb, J.A. Foster, E.D. de Jong, H. Lipson, X. Llora, S. Mancoridis, M. Pelikan, G.R. Raidl, T. Soule, A.M. Tyrrell, J.-P. Watson, E. Zitzler (2005) pp. 1729–1736, available online at http://www.cs.bham.ac.uk/~wbl/biblio/gecco2005/docs/p1729.pdf
43.168 J.H. Moore, C.S. Greene, P.C. Andrews, B.C. White: Does complexity matter? Artificial evolution, computational evolution and the genetic analysis of epistasis in common human diseases. In: *Genetic Programming Theory and Practice Vol. VI*, ed. by R.L. Riolo, T. Soule, B. Worzel (Springer, Berlin, Heidelberg 2008) pp. 125–145, Chap. 9
43.169 S. Gustafson: An Analysis of Diversity in Genetic Programming, Ph.D. Thesis (School of Computer

Science and Information Technology, University of Nottingham, Nottingham 2004), available online at http://www.cs.nott.ac.uk/~smg/research/publications/phdthesis-gustafson.pdf
43.170 G.S. Hornby: A steady-state version of the age-layered population structure EA. In: *Genetic Programming Theory and Practice*, Vol. VII, Genetic and Evolutionary Computation, ed. by R.L. Riolo, U.-M. O'Reilly, T. McConaghy (Springer, Ann Arbor 2009) pp. 87–102, Chap. 6
43.171 J.C. Bongard: Coevolutionary dynamics of a multi-population genetic programming system, Lect. Notes Comput. Sci. **1674**, 154 (1999), available online at http://www.cs.uvm.edu/~jbongard/papers/s067.ps.gz
43.172 I. Dempsey, M. O'Neill, A. Brabazon: *Foundations in Grammatical Evolution for Dynamic Environments*, Studies in Computational Intelligence, Vol. 194 (Springer, Berlin, Heidelberg 2009), available online at http://www.springer.com/engineering/book/978-3-642-00313-4
43.173 J. Doucette, P. Lichodzijewski, M. Heywood: Evolving coevolutionary classifiers under large attribute spaces. In: *Genetic Programming Theory and Practice Vol. VII*, ed. by R.L. Riolo, U.-M. O'Reilly, T. McConaghy (Springer, Berlin, Heidelberg 2009) pp. 37–54, Chap. 3
43.174 S. Luke: http://cs.gmu.edu/~eclab/projects/ecj/
43.175 S. Luke: *The ECJ Owner's Manual – A User Manual for the ECJ Evolutionary Computation Library*, 0th edn. online version 0.2 edition, available online at http://www.cs.gmu.edu/~eclab/projects/ecj/docs/manual/manual.pdf
43.176 D.W. Dyer: https://github.com/dwdyer/watchmaker
43.177 E. Hemberg, M. O'Neill: http://ncra.ucd.ie/Site/GEVA.html
43.178 M. O'Neill, E. Hemberg, C. Gilligan, E. Bartley, J. McDermott, A. Brabazon: GEVA: Grammatical evolution in Java, SIGEVOlution **3**(2), 17–22 (2008), available online at http://www.sigevolution.org/issues/pdf/SIGEVOlution200802.pdf
43.179 J. Dréo: http://eodev.sourceforge.net/
43.180 G. Squillero: http://www.cad.polito.it/research/Evolutionary_Computation/MicroGP/index.html
43.181 M. Schillaci, E.E. Sanchez Sanchez: A brief survey of μGP, SIGEvolution **1**(2), 17–21 (2006)
43.182 G. Squillero: MicroGP – an evolutionary assembly program generator, Genet. Program. Evol. Mach. **6**(3), 247–263 (2005), Published online: 17 August 2005.
43.183 C. Gagné, M. Parizeau: http://beagle.sourceforge.net/
43.184 C. Gagné, M. Parizeau: Open BEAGLE A C++ framework for your favorite evolutionary algorithm, SIGEvolution **1**(1), 12–15 (2006), available online at http://www.sigevolution.org/2006/01/issue.pdf
43.185 S. Silva: http://gplab.sourceforge.net/
43.186 F.M. De Rainville, F.-A. Fortin: http://code.google.com/p/deap/
43.187 L. Spector: http://hampshire.edu/lspector/push.html
43.188 R. Poli: http://cswww.essex.ac.uk/staff/rpoli/TinyGP/
43.189 E. Hemberg, J. McDermott: http://code.google.com/p/ponyge/
43.190 H. Lipson: http://creativemachines.cornell.edu/eureqa
43.191 M.E. Kotanchek, E. Vladislavleva, G.F. Smits: http://www.evolved-analytics.com/
43.192 P. Nordin: http://www.rmltech.com/
43.193 P. Nordin, W. Banzhaf, F.D. Francone: Efficient evolution of machine code for CISC architectures using instruction blocks and homologous crossover. In: *Advances in Genetic Programming*, Vol. 3, ed. by L. Spector, W.B. Langdon, U.-M. O'Reilly, P.J. Angeline (MIT, Cambridge 1999) pp. 275–299, Chap. 12, available online at http://www.aimlearning.com/aigp31.pdf
43.194 W.B. Langdon: http://www.cs.bham.ac.uk/~wbl/biblio/
43.195 W.B. Langdon: http://www.cs.ucl.ac.uk/staff/W.Langdon/homepages.html
43.196 Genetic Programming Yahoo Group: http://groups.yahoo.com/group/genetic_programming/
43.197 J. McDermott, D. White: http://gpbenchmarks.org
43.198 J. McDermott, D.R. White, S. Luke, L. Manzoni, M. Castelli, L. Vanneschi, W. Jaśkowski, K. Krawiec, R. Harper, K. De Jong, U.-M. O'Reilly: Genetic programming needs better benchmarks, Proc. GECCO 2012, Philadelphia (2012)
43.199 R. Poli, W.B. Langdon, N.F. McPhee: *A Field Guide to Genetic Programming* (Lulu, Raleigh 2008), Published via http://lulu.com and available at http://www.gp-field-guide.org.uk (With contributions by J. R. Koza)
43.200 S. Luke: *Essentials of Metaheuristics*, 1st edn. (Lulu, Raleigh 2009), available online at http://cs.gmu.edu/~sean/books/metaheuristics/
43.201 W. Banzhaf, P. Nordin, R.E. Keller, F.D. Francone: *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and Its Applications* (Morgan Kaufmann, San Francisco 1998), available online at http://www.elsevier.com/wps/find/bookdescription.cws_home/677869/description#description
43.202 W.B. Langdon, R. Poli: *Foundations of Genetic Programming* (Springer, Berlin, Heidelberg 2002), available online at http://www.cs.ucl.ac.uk/staff/W.Langdon/FOGP/
43.203 H. Iba, Y. Hasegawa, T. Kumar Paul: *Applied Genetic Programming and Machine Learning*, CRC Complex and Enterprise Systems Engineering (CRC, Boca Raton 2009)
43.204 M. Sipper: *Evolved to Win* (Lulu, Raleigh 2011), available at http://www.lulu.com/