Genetic Programming

James McDermott and Una-May O'Reilly
Evolutionary Design and Optimization Group,
Computer Science and Artificial Intelligence Laboratory,
Massachusetts Institute of Technology,
Cambridge, Massachusetts, USA.
{jmmcd, unamay}@csail.mit.edu

March 29, 2012

Contents

Chapter 1

Genetic Programming

Genetic programming is the subset of evolutionary computation in which the aim is to create an executable program. 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 hot topics of research and point to successful applications.

1.1 Introduction

There have been many attempts to artificially emulate human intelligence, from symbolic artificial intelligence to connectionism, to sub-cognitive approaches like behavioural artificial intelligence and statistical machine learning, and domain-specific achievements like Google search and the autonomous car. Darwinian evolution 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 several forms of intelligence very different from itself, including our own.

The principles of evolution—biased selection and inheritance with variation—serve as inspiration for the field of evolutionary computation, an adaptive learning and search approach which is general-purpose, applicable even with "black-box" performance feedback, and "embarrassingly parallel". In evolutionary computation "individuals" are evaluated for fitness, good ones are selected as parents, and new ones are created by inheritance with variation. See Figure 1.1.

Genetic programming is the subset of evolutionary computation in which knowledge and behaviour

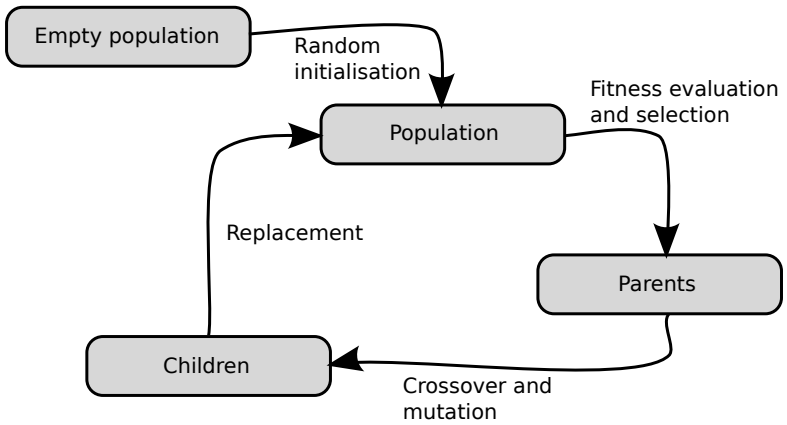


Figure 1.1: The fundamental loop of evolutionary computation

are evolved, represented by programs. The key distinguishing feature is that of evolving in an *executable* representation. A program's fitness is evaluated by executing it to see what it does. New, syntactically correct programs are created by inheriting material from their parents and by varying it.

Genetic programming is very promising, because programs are so general. A program can generate any other data structure: numbers, strings, lists, dictionaries, sets, permutations, trees, and graphs [51, 57, 126]. A program can emulate any model of computation, including Turing machines, cellular automata, neural networks, grammars, and finite state machines. [FIXME [128] for Turing machines?]

A program can be a data regression model [47] or a probability distribution **TODO: cite**. It can express the growth process of a plant [103], the gait of a horse [77], or the attack strategy of a group of lions [34]; it can model behaviour in the Prisoner's Dilemma [15] or play chess [33], Pacman [26], or a car-racing game [130]. A program can generate designs for physical objects, like a space-going antenna [60], or plans for the organisation of objects, like the layout of a manufacturing facility [24]. A program can implement a rule-based expert system for medicine [7], a scheduling strategy for a factory [35], or an exam timetable for a university [3]. A program can recognise speech [11], filter a signal [18], or smooth the raw output of a brain-computer interface [101]. It can generate a piece of abstract art [118], a 3D architectural model [86], or a piece of piano music [13].

A program can take input from a random number generator. A program can interface with natural or man-made sensors and actuators in the real world, so it can both act and react. It can interact with a user or with remote sites over the network [137]. It can also introspect and copy or modify itself [122]. If true artificial intelligence is possible, then a program can be intelligent [40]. **TODO: add citations**

Genetic programming exists in many different forms which differ (among other ways) in their executable representation. As in programming "by hand", genetic programming works more readily if it is allowed to create programs without a pre-stipulated length. Programs are also generally hierarchical in some sense. They often have statement or control nesting. Genetic programming works more readily if given the flexibility of exploring hierarchical programs. These representation properties (variable length and hierarchical structure) raise a very different set of technical challenges for genetic programming compared to typical evolutionary computation.

In each form of genetic programming, the term "program" may have a different meaning. We take a broad view: 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 genetic programming, 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 program code. This double meaning echoes that of DNA, which is both data and code in the same sense.

1.2 History

GP has a surprisingly long history, dating back to very shortly after John Von Neumann's 1945 description of the stored-program architecture [136] and the 1946 creation of ENIAC [30], sometimes regarded as the first general-purpose computer. In 1948 Alan Turing stated the aim of machine intelligence and recognised that evolution might have something to teach us in this regard:

> 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.
>
> —Turing, "Intelligent Machinery" [133]

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

> We have [...] divided our problem [automatic programming] into two parts. The child-programme [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
>
> —Turing, "Computing Machinery and Intelligence" [132].

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 automatic programming and machine intelligence. In 1959 Arthur Samuel wrote that the aim was to be able to "Tell the computer what to do, not how to do it" [109]. An important early attempt at implementation of automatic programming was the "learning machine" of R. M. Friedberg [22].

In 1963 John McCarthy summarised [65] 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 behaviour, density of good programs in the search space, sufficiency of the search space, appropriate fitness evaluation, and self-organised modularity. Many of these remain open issues in GP [85].

Lawrence J. Fogel's 1960s work using finite state machines as the representation for *evolutionary programming* can probably be called the first successful GP implementation. A detailed history is available in David Fogel's 2006 book [20].

One noticeable omission from Turing's description (above) is the idea of crossover, though one could argue that by "mutations" he intended to include all methods of genetic variation. Either way, the importance of crossover as a mechanism in both Darwinian evolution and EC raises an interesting question of primacy: who was the first to explicitly describe a crossover mechanism in program search? This honour also appears to go to Fogel, who in 1967 assessed the utility of crossover operators in the finite state machine representation [19]. However, more typically EP did not use crossover.

In the 1980s, inspired by the success of GAs and LCS, several authors experimented with hierarchically-structured and program-like representations. Smith [120] proposed a representation of a variable-length list of rules which could be used for program-like behaviour such as maze navigation and poker. Cramer [12] was the first to use a tree-structured representation and appropriate operators. A simple proof of concept, it successfully evolved a multiplication function in a simple custom language. Schmidhuber [111] describes a GP system with the possibility of Turing-completeness, though the focus is certainly on meta-learning aspects. Fujiki and Dickinson [23] generated Lisp code for the prisoner's dilemma, and Bickel and Bickel [6] used a GA to create variable-length lists of rules, each of which had a tree structure. Another approach was also successfully used by Ray [105]. 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 John Koza [47]. In addition to the abstract syntax tree representation (see Figure 1.3), the key innovations were probably subtree crossover (see Figure 1.3(b)) and the description and set-up of many test problems. In this and later research [48, 51, 52] 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. Mutation was minimised 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 authors began to work in the field, new types of GP were developed (see Section 1.3), successful applications appeared (see Section 1.4), key research topics were identified (see Section 1.5), further books were written, and conferences and journals were established (see Section 1.6).

Table 1.1: Acronyms for research fields and techniques. **TODO: can we delete this?**

Field/technique	acronym
Artificial intelligence	AI
Machine learning	ML
Automatic programming	AP
Evolutionary computation	EC
Inductive programming	IP
Genetic programming	GP
Evolutionary programming	EP
Evolutionary strategies	ES
Differential evolution	DE
Genetic algorithm	GA
Machine learning	ML
Particle swarm optimisation	PSO
Ant Colony Optimisation	ACO
Strongly-Typed GP	STGP
Cartesian GP	CGP
Linear GP	LGP
Standard GP	StdGP
Simulated annealing	SA
Hill climbing	HC
Probabilistic graphical model	PGM
Neural network	NN
Support vector machine	SVM
Learning classifier systems	LCS

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 (http://www.genetic-programming.org/hc2011/combined.html) again helps to demonstrate to researchers and clients outside the field of GP that it is powerful and general-purpose.

1.3 Taxonomy: Upward and Downward from GP

In this section we present a taxonomy which firstly places GP in the context of the broader fields of evolutionary computation, machine learning, and artificial intelligence. It then classifies GP techniques according to their representations and their population models. The acronyms we use throughout are set out in Table 1.1.

1.3.1 Upward

GP is a type of EC, which is a type of ML, which is itself a subset of the broader field of AI. Carbonell et al. [9] 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. They also classify ML techniques according to their *representation*—we deal with this issue in the next section.

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-

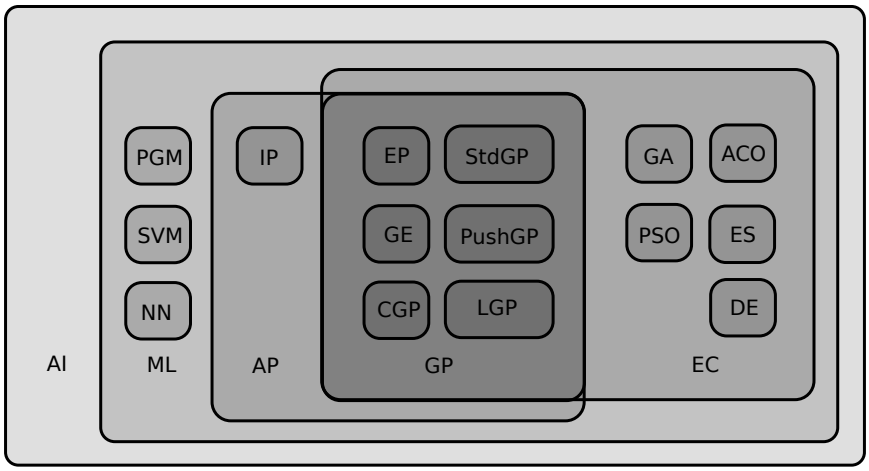


Figure 1.2: A taxonomy of AI, EC, and GP.

driven source generation, then generation techniques such as Universal Modelling Language (UML), to the ambitious aim of creating software directly from a natural-language English specification. "Automatic programming has been a goal of computer science and artificial intelligence since the first programmer came face to face with the difficulties of programming. As befits such a long-term goal, it has been a moving target—constantly shifting to reflect increasing expectations." [106]. We interpret AP to mean creating software by specifying *what to do* rather than *how to do it*. While GP clearly fits into this category, other non-evolutionary techniques do also. A good example is 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 specifications. In contrast, GP is concerned mostly with programs which are syntactically correct, but behaviourally suboptimal. GP uses heuristic methods to create programs and then test them against the specifications. Nevertheless IP (`http://www.inductive-programming.org/`) may serve as a useful source of inspiration and benchmarking for GP practitioners.

1.3.2 Downward

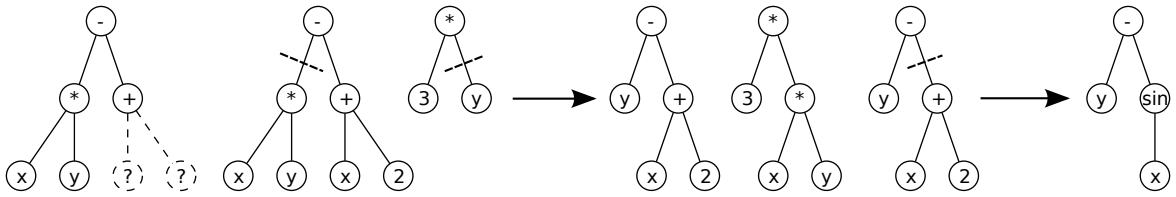
It is traditional to divide EC into four main sub-fields: ES, EP, GA, and GP. In this view, ES is chiefly characterised by real-valued optimisation and self-adaptation of algorithm parameters; EP by a finite state machine representation (later generalised) and the absence of crossover; GA by the bitstring 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 PushGP, GE, CGP, STGP, and several others treated below. In the following we classify GP algorithms according to their *representation* and according to their *population model*: see Fig. 1.2.

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. An example is 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 to RNA to mRNA to codons to proteins and finally to cells. The choice between direct and indirect representations also affects the shape of the fitness landscape (see Section 1.5.2). In the following we present a non-exhaustive selection of the main representations used in GP, in each case describing the three key operators: initialisation, 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 accept zero or more arguments of the same type and return a single value of the same type. Trees can be initialised by recursive random growth starting from a null node. StdGP uses parameterized initialisation methods that diversify the size and structure of initial trees. Fig. 1.3(a) shows a tree in the process of initialisation. Trees can be crossed-over by cutting and swapping the subtrees rooted at randomly-chosen nodes, as shown in Fig. 1.3(b). They can be mutated by cutting and regrowing from the subtrees of randomly-chosen nodes, as shown in Fig. 1.3(c). Another mutation operator, HVL-Prime, is shown in Fig. 1.10. 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 is *strongly-typed GP* [75, 145], in which nodes can have different input and output types, and all the above operations must respect these types.



(a) Initialisation (b) Crossover (c) Mutation

Figure 1.3: The StdGP representation is an abstract syntax tree. The expression that will be evaluated in the 2nd 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*.

Executable graph representations A natural generalisation 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) [91] 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) [74] uses an integer-array genome and a mapping process to produce the graph. Each chunk 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. See Fig. 1.4. *Neuro-evolution of augmented topologies* (NEAT) [125] 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 can use simple array-oriented (GA-style) initialisation, crossover, and mutation operators (subject to some constraints).

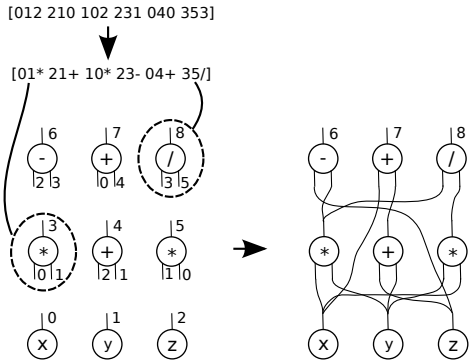


Figure 1.4: Cartesian GP. An integer-array genome is divided into chunks: in each chunk the last integer specifies a function (top-left). Then one node is created for each input variable (x, y, z) and for each genome chunk. Nodes are arranged in a grid and outputs are indexed sequentially (bottom-left). The first elements in each chunk 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.

Evolutionary programming The original incarnation of *evolutionary programming* (EP) [21] also uses graphs, but in this case the model of computation is the finite state machine rather than the executable functional graph (see Fig. 1.5). In a typical implementation [21], 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. Crossover is not used. **TODO: I can't find any explanations of initialisation or crossover in FSM-style EP!**

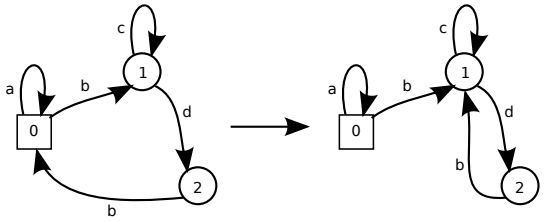


Figure 1.5: EP representation: finite state machine. In this example, a mutation changes a state transition.

Grammatical GP ¡¡¡¡¡¡¡ .mine The search space of *Grammatical GP* is a non-deterministic context-free grammar (CFG) [71]. Often the CFG defines a subset of the valid programs in a programming language such as C or Python which is then compiled or interpreted, or in a Lisp-like syntax which is interpreted. In one early system [138], 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 non-terminal, and crossover is constrained to exchange subtrees whose roots are identical non-terminals. 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* (GE) [83] instead uses an integer-array genome. Initialisation, 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 among the applicable production choices at each step of the derivation process. Fig. 1.6 shows a simple grammar, integer genome, derivation process, and derivation tree. At each step of the derivation process, the left-most non-terminal in the derivation is re-written. 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. ======= In *grammatical GP* [71] 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 non-deterministic CFG. The aim is to find a good program in that space. Often the CFG defines a subset of a programming language such as Lisp, C or Python which is then compiled or interpreted. The advantages of using a CFG are that it allows a fine-grained definition of multiple data-types and the imposition of 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 express restrictions on the ways program expressions are combined, for example making the system "dimensionally aware" [44, 104]. A grammatical GP system can 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 [138], 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 non-terminal, and crossover is constrained to exchange subtrees whose roots are identical non-terminals. 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* (GE) [83] instead uses an integer-array genome. Initialisation, 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 among the applicable production choices at each step of the derivation process. Fig. 1.6 shows a simple grammar, integer genome, derivation process, and derivation tree. At each step of the derivation process, the left-most non-terminal in the derivation is re-written. 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. ¿¿¿¿¿¿¿ .r3592 Although successful

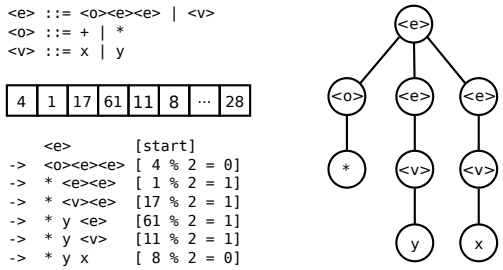


Figure 1.6: Grammatical GP representation: grammar, genome, derivation process, and derivation tree.

and widely-used, GE has also been criticised for the disruptive effects of its operators. Another system, *tree adjoining grammar-guided genetic programming* (TAG3P) has also been used successfully [36]. 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 [36]. However, it is a more complex representation.

Another common alternative approach, surveyed by Shan et al. [114], uses probabilistic models over grammar-defined spaces, rather than direct evolutionary search.

Linear GP In *Linear GP*, the program is a list of instructions to be interpreted sequentially. In order to achieve complex functionality, a set of registers are used. Instructions can read from or write to the registers. Several registers, which may be read-only, are initialised 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, a linear GP program can be seen as having a graph structure. A typical implementation is that of [8]. 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 (see Fig. 1.7). It also allows conditional statements and looping. It explicitly recognises the possibility of non-functioning code, or *introns*. Since there are no syntactic constraints on how multiple instructions may

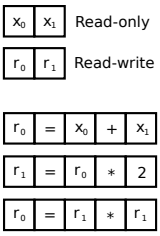


Figure 1.7: Linear GP representation: a list of register-oriented instructions. In this example program of three instructions, r_0 is the output register, and the formula $4(x_0 + x_1)^2$ is calculated.

be composed together, initialisation 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, and crossover can be performed over the two parents' list structures. More complex possibilities also exist. An interesting alternative is the implementation of Poli and McPhee [96], which takes advantage of the linear representation to use an n-gram based estimation of distribution algorithm. Initialisation is again by random generation, but reproduction is implemented by observing the frequency of n-grams of instructions among highly-fit programs, and then generating new programs with the same frequencies.

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 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 initialisation, mutation, and crossover as simple list-based operations [90]. 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 [54]. A more sophisticated type of stack-based GP is *PushGP* [123], 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. [112] automatically repaired programs written in Java byte code and in x86 assembly. Orlov and Sipper [89] evolved programs such as trail navigation and image classification *de novo* in Java byte code. This work made use of a specialised crossover operator which performed automated checks for compatibility of the parent programs' stack and control flow state. Nordin [79] 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.

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 [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 [131].

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 Arcuri and Yao [2], 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 PSO individuals tend to move towards the global best and towards the best point in their own history, but tend to avoid moving too close to other individuals. Although PSO and related methods such as DE are best applied in real-valued optimisation, their population models and operators can be abstracted and applied in GP methods also [76, 82].

Finally, we come to **estimation of distribution algorithms** (EDAs). Here the idea is to create a population, select a sub-sample of the best individuals, model that sub-sample using a distribution, and then create a new population by sampling the distribution. This approach is particularly common in grammar-based GP [70], though it is used with other representations also [96, 61]. The modelling-sampling process could be regarded as a type of multi-parent 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.

1.4 Well Suited Domains and Noteworthy Applications of GP

Our introduction (Section 1.1) 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.

1.4.1 Symbolic regression

Symbolic regression is one of the most common tasks for which GP is used. It is used as a component in techniques like data modelling, clustering, and classification, for example in the modelling application outlined in Sect. 1.4.2. It can be seen as a generalisation of techniques such as linear or quadratic regression, but does not require *a priori* specification of the model as those techniques do.

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. 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. As in other machine learning methods, a subset of the data is withheld for testing purposes. The remaining n rows are termed the training set.

Typically StdGP is used. It is set up with a numerical "language" which includes arithmetic operators and numerical constants, sometimes functions like sinusoids and exponentials, and the input variables of the dataset. The operators and functions can be internal nodes of an abstract syntax tree and the constants and variables its leaf nodes. An initial population of StdGP trees is generated, each regarded as a model. To calculate the fitness of each model, each training point's explanatory variable is bound to the corresponding input variable of the model and the expression is executed. The output of the expression is the model's predicted response. This value is then comparable to the response of the training point. 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 minimised:

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

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

1.4.2 GP and Machine Learning

Like other machine learning 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 machine learning methods in circumstances where the form of the solution model is unknown *a priori*, see Figure 1.8, left. For the sensory evaluation problem described in [**?**], the authors use GP as an anchor of a ML framework, see Figure 1.8, right. Their 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 employs genetic programming symbolic regression and ensemble methods to generate multiple diverse explanations of assessor liking preferences with confidence information. It uses statistical techniques to extrapolate from the genetically evolved model ensembles to unobserved regions of the flavor space and to segment 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 (see [**?**] 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 hedonic response directionality when the important variables are varied, were grouped together to identify assessors who are driven by the same ingredient set, 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 constitue 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.

1.4.3 Software Engineering

At least three aspects of software engineering have been tackled with remarkable success by GP: bug-fixing [58], parallelisation [108, 140], and optimisation [139]. These three projects 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 have two key features in common: they 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. These techniques have also been proven on real-world problems.

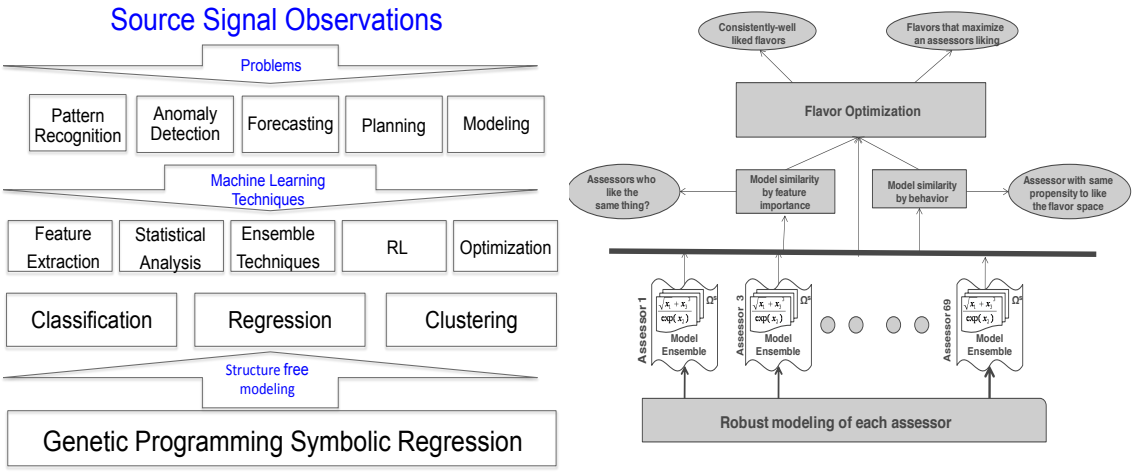


Figure 1.8: Genetic programming symbolic regression is unique and useful as a machine learning technique because it obviates the need to define the structure of a model prior to training.

Le Goues et al. [58] show that automatically fixing software bugs is a problem within the reach of GP. They describe a system called *GenProg* for automatically fixing several classes of bugs in a generic way. It operates on C source code taken from open-source projects. GenProg 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 sub-expressions, and they are constrained to alter only lines which are exercised by the failing test cases. Again, this helps to reduce the search space size. The original positive test cases 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 success 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 are demonstrated to be rather robust: some even generalise to fixing related bugs which were not explicitly encoded in the test suite.

Ryan [108] describes a system, *Paragen*, which automatically re-writes 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. In *Paragen II*, correctness of the new programs is guaranteed by 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. Paragen II is thus a meta-level GP algorithm. The work of Williams [140] was in some ways similar to Paragen I.

A third goal of software engineering is optimisation of existing code. White et al. [139] tackle this task using a multi-objective optimisation 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 "non-obvious" optimisations, i.e. ones which cannot be found by optimising compilers. A population of test cases is coevolved with the population of programs.

Other GP-based software engineering work includes that of O'Keeffe and Ó Cinnéide [81] on automated refactoring, in which a large set of possible refactoring steps are provided as types of mutation. A survey of the broader field of *search-based software engineering* is given by Harman [31].

1.4.4 Art, Music, and 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.

One of the first examples of GP design was the synthesis of analog electrical circuits by Koza et al. [50]. This work attacked the problem of automatically creating circuits to perform tasks such as a filter or an amplifer. 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 (see Section 1.3.2) and ADFs (see Section 1.5.3). Execution of the evolved program began with a trivial "embryonic circuit". GP program nodes, when executed, performed a actions such as altering the circuit topology or creating a new component. These nodes were parameterised 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.

Another significant success story was the space-going antenna evolved by Lohn et al. [60] for the NASA 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 labour 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 3D turtle graphics system. It relied on the idea of "state", representing the current position and orientation of a "turtle". Commands included *forward* which moved the state 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 **TODO: did this antenna actually make it to space?**.

There have also been successes in the fields of graphical art, 3D 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* [14] and has been used in other forms of EC also [127]. Early successes were those of Todd and Latham [129], who created pseudo-organic forms, and Sims [118] who created abstract art. An excellent overview of evolutionary art is provided by Lewis [59].

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 [69].

As a result, many authors have tried to use representations which take advantage of forms of *re-use*. Although re-use is also an aim in non-aesthetic GP (see Section 1.5.3), the hypothesis that good solutions will tend to involve re-use, 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 pre-defined, 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 re-used many times, and all parts of the work are felt to be coherent. The earliest example of such work was that of Sims [118], who created fascinating graphical art (a 2D grid) and some animations (a 3D grid of 2 spatial dimensions and 1 time dimension). The paradigm was later brought to a high degree of artistry by Hart [32]. The same generative idea, now with a 1D grid, was used by Hoover et al. [38], Shao et al. [115] and McDermott and O'Reilly [67] to produce music as a function of time, and by Clune and Lipson [10] to produce 3D designs.

Other successful work has used different approaches to re-use. *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 sub-strings. Interpreting this string as a program can then yield highly patterned graphics [66], artificial creatures [39], and music [144]. Grammars have also been used in 3D and architectural design, both in a modified L-system form [86] and in the standard GE form [68]. Although not explicitly grammatical, the *Ossia* system of Dahlstedt [13] again uses a recursive structure (GP trees with recursive pointers) to impose re-use and a natural, *gestural* quality on short pieces of art-music.

1.5 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.

1.5.1 Bloat

Since the vast majority of GP-type problems are not amenable to *a priori* specification of the exact size of the solution, variable-length representations are needed. Program size is constrained to be nonnegative, but it is natural to consider the possibility that some programs will become very large. It might be expected that selection pressure would effectively guide the population towards sizes appropriate to the problem, and indeed this is sometimes the case. However it has also been observed that in some circumstances, for many different representations and problems, programs grow over time *without* apparent fitness improvements. *Bloat* is the name given to this phenomenon, and is defined as "program growth without (significant) return in terms of fitness." [99, p. 101]. Since the time complexity for the evaluation of a GP program is generally proportional to its size, this slows the GP run down unnecessarily. The outcome has been well described by Luke and Panait [64] as "a kind of Zeno's paradox": each generation becomes slower than the last, and it can appear unprofitable to continue the run.

A second drawback of uncontrolled growth in program size is that the eventual solution may be so large and complex that is unreadable, negating a key advantage of symbolic methods like GP. Overly-large programs also tend to generalise less well than parsimonious ones.

Clearly, then, bloat is a significant obstacle to successful GP. It has become 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 summarised by Luke and Panait [64] and Silva et al. [117]. A fundamental idea is that adding material to a GP tree is more likely than removing material from a tree to lead to a fitness improvement. The *intron theories* include: *hitchhiking*, where non-effective code is carried along by virtue of being attached to useful code; *defense against crossover*, which suggests that large amounts of non-effective code give a selection advantage later in GP runs when crossover is likely to highly destructive of good, fragile programs; and *removal bias*, which is the idea that it is harder for GP operators to remove exactly the right (i.e. non-effective) code than it is to add more. The *fitness causes bloat* theory [55] is different. When fitness improvements are hard to find, most new individuals are at best fitness-neutral with respect to their parents, especially for discrete-valued fitness functions. However since there are many more programs with the same functionality at larger sizes than at smaller, there is a drift towards larger programs. 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 towards growth. Finally, the *crossover bias* theory [16] explains the effect of bloat by firstly ignoring selection, and concentrating on the effects of the crossover operator in isolation. Simply put, after many crossovers, a population will tend towards a limiting distribution of tree sizes [53]. In this distribution, 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 the 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 [47].

Another simple method is *parsimony pressure*, i.e. a fitness penalty imposed on overly-large individuals. This implicitly or explicitly assumes that fitness is commensurable with size: the magnitude of the punishment effectively establishes an "exchange rate" between the two. Luke and Panait [64] found that parsimony pressure was surprisingly effective across problems and across a wide range of exchange rates.

The choice of a *de facto* exchange rate can be avoided. One can define a multi-objective algorithm in which one of the objectives is fitness and the other program length or complexity, for example ParetoGP [121]. The correct definition for complexity in this context is itself an interesting research topic [135, 134]. Alternatively the pressure can be moved into the selection phase of the algorithm instead of the fitness evaluation phase, using the *double tournament* method [64]. 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 [41].

Another simple but surprisingly effective method is *Tarpeian bloat control* [102]. The basic idea is that individuals which are larger than average receive, with a certain probability, a constant, punitively bad fitness instead of being evaluated. It can therefore be seen as a variation on the *parsimony pressure* method. The key advantage of the method is that the large, unlucky individuals are not evaluated, and so a huge amount of time can be saved and devoted to running more generations (as in [64]). 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 be set adaptively in a later variation of the method [102]. The Tarpeian method is theoretically well-motivated and has achieved good results on benchmarks and real-world problems.

Another important strand of research into bloat control is known as *operator length equalisation*. It is motivated by the *crossover bias* theory. At each generation a distribution of program sizes is formed. In the distribution the frequency of programs with a particular size depends on the fitness of existing programs of that size. This distribution is then used to control which new individuals will be accepted into the population. Individuals of a particular size are accepted if the capacity of the distribution at that size has not been exceeded, *or* if they are exceptionally fit for that size. Individuals whose size places them outside the distribution entirely can be accepted, again, if they are exceptionally fit. Operator length equalisation is thus a dynamic method. A *mutation-based* variation of the method does not reject individuals which fall outside the desired distribution, but instead mutates them using directed mutations to become smaller or larger as needed. This tends to save CPU time. The method has enjoyed significant success. Although it imposes a programming burden larger than many other methods, recent work has also shown that in most practical cases the desired size distribution is effectively flat [116]. This suggests that a simpler implementation may be possible.

Although some of the above theories and methods are representation-specific, it is also possible to think about bloat in a representation-free way [4]. Some authors have argued that the choice of GP representation can avoid the issue of bloat [73].

Some authors also aim to avoid the problem of bloat rather than tackle it head-on. One possibility is to make GP fitness evaluation more computationally efficient by using word-level parallelism in the CPU [97], optimising it for GPU [54], producing and evaluating individuals *lazily* [93], or caching the results of subtrees [43].

In summary, researchers including Luke and Panait [64], Poli et al. [102], Miller [73] and Silva et al. [117] have effectively "declared victory" in the fight against bloat. However, their techniques have not yet become *de rigeur* for new GP research and benchmark experiments.

1.5.2 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's 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 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 three influential methods of theoretical analysis and very briefly describe them and their results: schema-based analysis, Markov chain modeling, and runtime complexity. We also include a brief introduction to the No Free Lunch Theorem to explain its implications for GP.

In schema-based analysis the search space is conceptually partitioned into hyperplanes (a.k.a schemas) which represent sets of partial solutions. There are numerous ways to do this and, as a consequence, multiple schema definitions have been proposed [1, 49, 88, 92, 107]. 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 by [94, 95] for most types of crossover. These alternatively depend upon explicitizing the effects and the mechanisms of schema creation. This is insightful, however, tracking schema equations in actual GP population dynamics is infeasible. As well, while schema theorems predict 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 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 [98]. Another interesting Markov chain-based 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 influence studied in a semantic building block analysis by [72]. 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.

In a nutshell, the No Free Lunch Theorem [141] proves that, averaged over all problem instances, no algorithm outperforms another. Follow-up NFL analysis [113, 143] 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 differerent programs have the same functionality while program output functionality is not uniformly distributed across syntax trees. Second, a geometric argument has shown [100], 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.

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 [17]. The authors study the runtime of hill climbing GP algorithms which use a mutation operator called HVL-Prime, see Figures 1.9 and 1.10. Several of these simplified GP algorithms were analyzed on two separable model problems, ORDER and MAJORITY introduced in [28]. 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 gives 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[29]. It reflects a general (and thus weak) property required of GP solutions: a solution must have correct functionality (by evolving an aggregation of sub-solutions) 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, [46], regarding mutation arose from analysis of the *Max* problem [27] 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 $O(nlogn)$ if the mutations are biased to replace a random leaf with distance d from the root with probability 2^{-d}.

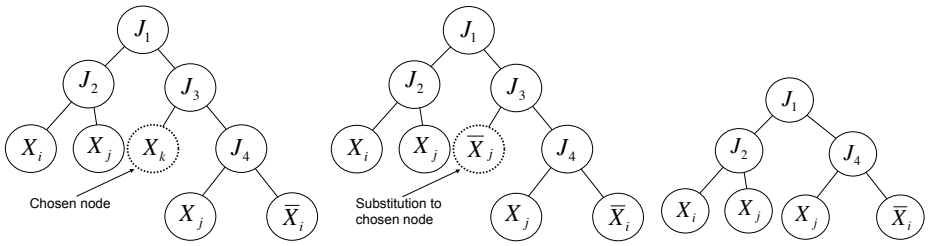


Figure 1.9: HVL-Prime: Left: Parse tree before substitution, deletion, Center: Result of substitution, Right: Result of deletion

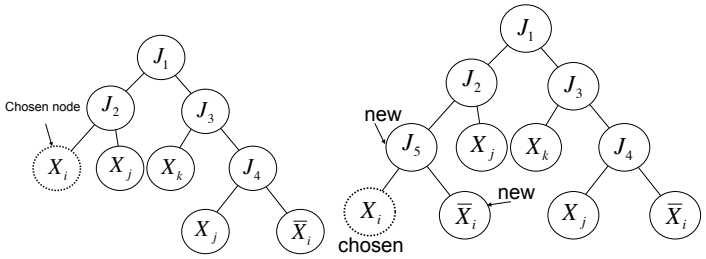


Figure 1.10: HVL-Prime: Left: Parse tree before insertion, Right: Result of insertion

Runtime analyses have also considered parsimony pressure and multi-objective GP algorithms for generalizations of ORDER and MAJORITY [78].

Other theoretical avenues of investigation which are noteworthy, but which can not be covered here, include fitness landscape and problem difficulty as well as the analysis and amelioration of bloat. GP algorithms have also been studied in the PAC learning framework [45].

1.5.3 Modularity

Modularity. ADFs, ADMs, etc. Other approaches. Hornby on modularity/regularity/hierarchy. Olson's approach to abstraction/ADFs in ADATE [37]. [FIXME Please mention re-use as a motivation/method since I refer here from the art section. Bentley's embryogeny stuff?]

1.5.4 Open-ended GP

Segue from modularity to here:

Automatically-defined iterations, loops, recursions, and stores: [51]. Architecture-altering: "The idea of architecture altering operations was extended to the ex- tremely general Genetic Programming Problem Solver (GPPS), which is described in detail in (Koza et al., 1999, part 4). This is an open ended system which combines a small set of basic vector-based primitives with the architecture altering operations in a way that can, in theory, solve a wide range of problems with almost no input required from the user other than the fitness function. The problem is that this open-ended system needs a very carefully constructed fitness function to guide it to a viable solution, an enormous amount of computational effort, or both. As a result it is currently an idea of more conceptual than practical value."—[Field Guide p. 51].

True GP as a research topic: In his first (?) book, Koza presents a high-level view of EC: we have done everything with data structures—meaning optimisation of bitstrings, continuous values, permutations, etc.—but not nearly enough with programs. But right after saying this, he writes four massive books which really don't deal with full-on programs, only mathematical formulae. Many people have worked to reclaim the vision of real genetic programming and more generally automated programming: Tina Yu, Lee Spector, Alex Agapitos, Simon Lucas and others. Harrington. Zippers – a method of copying/varying individuals. Autoconstructive evolution – all levels of abstraction (pop, ind, component) are a part of the program itself. Homoiconicity. In Push, programs can self-modify while they run via exec stack. We would like to be in that tradition also. We could call it *True GP*.

More examples:

J. R. Woodward, "Evolving Turing complete representations," in Proc. CEC, vol. 2. Dec. 2003, pp. 830837. [142]

open-ended, large-scale, modularity, cloud

From GPEM article: Open-ended evolution in GP Issue: Design an evolutionary system capable of continuously adapting and searching. Difficulty: Medium to Hard. There is difficulty in defining clear criterion for measuring success, and whether or not open-ended evolution is required by GP practitioners. Can we achieve open-ended evolution with GP, where, for example, a GP system forms the core part of a financial trading system required to continuously learn without being switched off? Would it be possible that that same system can adapt over time to a completely different problem? Notions of computational evolution can again provide a bridge between biology and EC, as exemplified by recent work of Moore and co-workers [87]. Essential ingredients of open-ended evolution are (i) a dynamically changing fitness landscape, (ii) availability of co-evolutionary processes, (iii) search with contin- uously injected randomness. The latter point is important. How can a GP system be open to randomness and at the same time stabilize its best adapted solutions? Tiered systems whose basic layer consist of random solutions that are being promoted to higher layers based on their performance seems to be a feasible way to achieve this goal. Systems based on fitness layers or on age layers have shown considerable success in this context [45, 46].

1.6 Practicalities

1.6.1 Conferences and Journals

Several conferences provide friendly venues for the publication of new GP work. The ACM *Genetic and Evolutionary Computation Conference* (GECCO) alternates annually between North America and the rest of the world and includes high-quality GP tracks. *EuroGP* is held annually in Europe as the main event of *Evo**, and focusses 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, Michigan, USA and provides a focussed forum for GP discussion. Although previously invitation-only, it has recently begun an open submission process. *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 small and theory-focussed conference, though not always including GP material.

The journal most specialised to the field is probably *Genetic Programming and Evolvable Machines* (published by Springer). The September 2010 10-year anniversary issue included several review articles which are a useful view of the literature. *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*.

1.6.2 Software

A great variety of GP software is available. Most researchers will be interested only in open-source software: in this area, as in others, the quality varies widely. A useful rule of thumb is to avoid packages which have not had recent updates or releases (since unmaintained code is likely less flexible and less reliable), or which use colourful and non-standard terminology in their documentation and source code (since it may indicate that the authors do not have significant experience in the field). We will recommend only a few packages.

One of the Java heavyweights is *ECJ* [63] http://cs.gmu.edu/~eclab/projects/ecj/. 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 very helpful mailing list. *Watchmaker* is another good-quality general-purpose system with excellent out-of-the-box examples https://github.com/dwdyer/watchmaker. *GEVA* [84] http://ncra.ucd.ie/Site/GEVA.html is another Java-based package, this time with support only for GE.

For users of C++ the situation is more difficult. Popular packages including *OpenBeagle* [25] http://beagle.sourceforge.net/ and *GPC++* appear to be unmaintained. *Evolutionary Objects* http://eodev.sourceforge.net/ and μGP http://www.cad.polito.it/research/Evolutionary_Computation/MicroGP/index.html [110, 124] are better bets.

Matlab users may be interested in GPLab http://gplab.sourceforge.net/, which implements standard GP, while DEAP http://code.google.com/p/deap/ provides implementations of several algorithms in Python. PushGP is available in many languages http://hampshire.edu/lspector/push.html.

Many more options are available—see for example the list maintained at http://en.wikipedia.org/wiki/Genetic_programming#Implementations. Two more systems are worth mentioning for their deliberate focus on simplicity and understandability. *TinyGP* http://cswww.essex.ac.uk/staff/rpoli/TinyGP/ and *PonyGE* http://code.google.com/p/ponyge/ 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* http://creativemachines.cornell.edu/eureqa is a high-quality and free-to-use tool containing a great deal of GP modelling technology.

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) http://www.evolved-analytics.com/ runs as a plugin to Mathematica. Its primary technology is the ParetoGP method [121], which gives the power of trading off program fitness against complexity, and forming ensembles of programs. It has useful tools for automatically dealing with ill-conditioned data and extracting information on variable importance. *Discipulus* (Register Machine Learning Technologies, Inc.) http://www.rmltech.com/ evolves machine code based on the ideas of Nordin et al. [80]. 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.

1.6.3 Resources and Further Reading

An indispensable resource for GP research is Bill Langdon's *GP Bibliography* http://www.cs.bham.ac.uk/~wbl/biblio/. In addition to its huge, regularly updated collection of BibTeX-formatted citations,

it has lists of researchers' homepages http://www.cs.ucl.ac.uk/staff/W.Langdon/homepages.html and co-authorship graphs.

The *GP mailing list* http://groups.yahoo.com/group/genetic_programming/ is also useful, though only sporadically active for anything other than announcements. Newcomers should beware that questions are generally ignored if they give the impression of asking for help as a first resort.

Many of the traditional GP benchmark problems have been criticised for being unrealistic in various ways. The lack of standardisation of benchmark problems also allows the possibility of cherry-picking of benchmarks. Effort is underway to bring some standardisation to the choice of GP benchmarks at the *GP Benchmarks wiki* http://groups.csail.mit.edu/EVO-DesignOpt/GPBenchmarks/.

Those wishing to read further have many good options. The *Field Guide* is a wonderful introduction and resource, walking the reader through simple examples, scanning vast amounts of the existing literature, and offering practical advice [99]. Luke's *Essentials of Metaheuristics* [62] also has an enjoyable, introductory style, but is broader in scope. Both are free to download. Other broad and introductory books include those by Fogel [20] and Banzhaf et al. [5]. More specialised books include those by Langdon and Poli [56] (coverage of theoretical topics), Langdon [57] (narrower coverage of GP with data structures), O'Neill and Ryan [83] (GE), Iba et al. [42] (GP-style machine learning), and Sipper [119] (games).

Acknowledgements

JMcD is funded by the Irish Research Council for Science, Engineering and Technology, co-funded by Marie Curie. U-MO'R is funded by FIXME.

Bibliography

[1] Lee Altenberg. Emergent phenomena in genetic programming. In Anthony V. Sebald and Lawrence J. Fogel, editors, *Evolutionary Programming — Proceedings of the Third Annual Conference*, pages 233–241, San Diego, CA, USA, 24-26 February 1994. World Scientific Publishing. ISBN 981-02-1810-9. URL http://dynamics.org/~altenber/PAPERS/EPIGP/.

[2] Andrea Arcuri and Xin Yao. A novel co-evolutionary approach to automatic software bug fixing. In Jun Wang, editor, *2008 IEEE World Congress on Computational Intelligence*, Hong Kong, 1-6 June 2008. IEEE Computational Intelligence Society, IEEE Press.

[3] Mohamed Bahy Bader-El-Den, Riccardo Poli, and Shaheen Fatima. Evolving timetabling heuristics using a grammar-based genetic programming hyper-heuristic framework. *Memetic Computing*, 1(3): 205–219, 2009. URL 10.1007/s12293-009-0022-y.

[4] W. Banzhaf and W. B. Langdon. Some considerations on the reason for bloat. *Genetic Programming and Evolvable Machines*, 3(1):81–91, March 2002. ISSN 1389-2576. doi: doi:10.1023/A:1014548204452. URL http://web.cs.mun.ca/~banzhaf/papers/genp_bloat.pdf.

[5] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, San Francisco, CA, USA, January 1998. ISBN 1-55860-510-X. URL http://www.elsevier.com/wps/find/bookdescription.cws_home/677869/description#description.

[6] Authur S. Bickel and Riva Wenig Bickel. Tree structured rules in genetic algorithms. In John J. Grefenstette, editor, *Genetic Algorithms and their Applications: Proceedings of the second International Conference on Genetic Algorithms*, pages 77–81, MIT, Cambridge, MA, USA, 28-31 July 1987. Lawrence Erlbaum Associates.

[7] Celia C. Bojarczuk, Heitor S. Lopes, and Alex A. Freitas. Genetic programming for knowledge discovery in chest-pain diagnosis. *IEEE Engineering in Medicine and Biology Magazine*, 19(4):38–44, July-August 2000. ISSN 0739-5175. URL http://ieeexplore.ieee.org/iel5/51/18543/00853480.pdf.

[8] Markus Brameier and Wolfgang Banzhaf. *Linear Genetic Programming*. Number XVI in Genetic and Evolutionary Computation. Springer, 2007. ISBN 0-387-31029-0. URL http://www.springer.com/west/home/default?SGWID=4-40356-22-173660820-0.

[9] J.G. Carbonell, R.S. Michalski, and T.M. Mitchell. An overview of machine learning. In J.G. Carbonell, R.S. Michalski, and T.M. Mitchell, editors, *Machine learning: An artificial intelligence approach*. I. Tioga Publ. Co, 1983.

[10] J. Clune and H. Lipson. Evolving three-dimensional objects with a generative encoding inspired by developmental biology. In *Proceedings of the European Conference on Artificial Life*, 2011. URL http://endlessforms.com.

[11] Markus Conrads, Peter Nordin, and Wolfgang Banzhaf. Speech sound discrimination with genetic programming. In Wolfgang Banzhaf, Riccardo Poli, Marc Schoenauer, and Terence C. Fogarty, editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391 of *LNCS*, pages 113–129, Paris, 14-15 April 1998. Springer-Verlag. ISBN 3-540-64360-5.

[12] Nichael Lynn Cramer. A representation for the adaptive generation of simple sequential programs. In John J. Grefenstette, editor, *Proceedings of an International Conference on Genetic Algorithms and the Applications*, pages 183–187, Carnegie-Mellon University, Pittsburgh, PA, USA, 24-26 July 1985. URL http://www.sover.net/~nichael/nlc-publications/icga85/index.html.

[13] Palle Dahlstedt. Autonomous evolution of complete piano pieces and performances. In *Proceedings of Music AL Workshop*, 2007.

[14] Richard Dawkins. *The Blind Watchmaker*. Longman Scientific and Technical, Harlow, England, 1986.

[15] Robert De Caux. Using genetic programming to evolve strategies for the iterated prisoner's dilemma. Master's thesis, University College, London, September 2001. URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/decaux.masters.zip.

[16] Stephen Dignum and Riccardo Poli. Generalisation of the limiting distribution of program sizes in tree-based genetic programming and analysis of its effects on bloat. In Dirk Thierens, Hans-Georg Beyer, Josh Bongard, Jurgen Branke, John Andrew Clark, Dave Cliff, Clare Bates Congdon, Kalyanmoy Deb, Benjamin Doerr, Tim Kovacs, Sanjeev Kumar, Julian F. Miller, Jason Moore, Frank Neumann, Martin Pelikan, Riccardo Poli, Kumara Sastry, Kenneth Owen Stanley, Thomas Stutzle, Richard A Watson, and Ingo Wegener, editors, *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 2, pages 1588–1595, London, 7-11 July 2007. ACM Press. URL http://www.cs.bham.ac.uk/~wbl/biblio/gecco2007/docs/p1588.pdf.

[17] Greg Durrett, Frank Neumann, and Una-May O'Reilly. Computational complexity analysis of simple genetic programming on two problems modeling isolated program semantics, 27 July 2010. URL http://arxiv.org/pdf/1007.4636v1. arXiv:1007.4636v1.

[18] Anna Esparcia-Alcazar and Ken Sharman. Genetic programming for channel equalisation. In Riccardo Poli, Hans-Michael Voigt, Stefano Cagnoni, Dave Corne, George D. Smith, and Terence C. Fogarty, editors, *Evolutionary Image Analysis, Signal Processing and Telecommunications: First European Workshop, EvoIASP'99 and EuroEcTel'99*, volume 1596 of *LNCS*, pages 126–137, Goteborg, Sweden, 28-29 May 1999. Springer-Verlag. ISBN 3-540-65837-8. URL http://www.iti.upv.es/~anna/papers/evoiasp99.ps.

[19] D.B. Fogel. Unearthing a fossil from the history of evolutionary computation. *Fundamenta Informaticae*, 35(1):1–16, 1998.

[20] D.B. Fogel. *Evolutionary computation: toward a new philosophy of machine intelligence*, volume 1. Wiley-IEEE Press, 2006.

[21] L.J. Fogel, P.J. Angeline, and D.B. Fogel. An evolutionary programming approach to self-adaptation on finite state machines. In *Proceedings of the fourth international conference on evolutionary programming*, pages 355–365, 1995.

[22] R.M. Friedberg. A learning machine: Part i. *IBM Journal of Research and Development*, 2(1):2–13, 1958.

[23] Cory Fujiki and John Dickinson. Using the genetic algorithm to generate lisp source code to solve the prisoner's dilemma. In John J. Grefenstette, editor, *Genetic Algorithms and their Applications: Proceedings of the second international conference on Genetic Algorithms*, pages 236–240, MIT, Cambridge, MA, USA, 28-31 July 1987. Lawrence Erlbaum Associates.

[24] Marcus Furuholmen, Kyrre Harald Glette, Mats Erling Hovin, and Jim Torresen. Scalability, generalization and coevolution – experimental comparisons applied to automated facility layout planning. In Guenther Raidl, Franz Rothlauf, Giovanni Squillero, Rolf Drechsler, Thomas Stuetzle, Mauro Birattari, Clare Bates Congdon, Martin Middendorf, Christian Blum, Carlos Cotta, Peter Bosman, Joern Grahl, Joshua Knowles, David Corne, Hans-Georg Beyer, Ken Stanley, Julian F. Miller, Jano van Hemert, Tom Lenaerts, Marc Ebner, Jaume Bacardit, Michael O'Neill, Massimiliano Di Penta, Benjamin Doerr, Thomas Jansen, Riccardo Poli, and Enrique Alba, editors, *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 691–698, Montreal, 8-12 July 2009. ACM. URL http://doi.acm.org/10.1145/1569901.1569997.

[25] Christian Gagné and Marc Parizeau. Open BEAGLE A C++ framework for your favorite evolutionary algorithm. *SIGEvolution*, 1(1):12–15, April 2006. URL http://www.sigevolution.org/2006/01/issue.pdf.

[26] Edgar Galvan-Lopez, John Mark Swafford, Michael O'Neill, and Anthony Brabazon. Evolving a Ms. PacMan controller using grammatical evolution. In Cecilia Di Chio, Stefano Cagnoni, Carlos Cotta, Marc Ebner, Aniko Ekart, Anna I. Esparcia-Alcazar, Chi-Keong Goh, Juan J. Merelo, Ferrante Neri, Mike Preuss, Julian Togelius, and Georgios N. Yannakakis, editors, *EvoGAMES*, volume 6024 of *LNCS*, pages 161–170, Istanbul, 7-9 April 2010. Springer. doi: doi:10.1007/978-3-642-12239-2_17.

[27] Chris Gathercole and Peter 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, 80 South Bridge, Edinburgh, EH1 1HN, UK, 1995. URL http://citeseer.ist.psu.edu/gathercole95max.html.

[28] David E. Goldberg and Una-May O'Reilly. Where does the good stuff go, and why? how contextual semantics influence program structure in simple genetic programming. In Wolfgang Banzhaf, Riccardo Poli, Marc Schoenauer, and Terence C. Fogarty, editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391 of *LNCS*, pages 16–36, Paris, 14-15 April 1998. Springer-Verlag. ISBN 3-540-64360-5. URL http://citeseer.ist.psu.edu/96596.html.

[29] D.E. Goldberg. *Genetic algorithms in search, optimization, and machine learning*. Addison-wesley, 1989.

[30] H.H. Goldstine and A. Goldstine. The electronic numerical integrator and computer (ENIAC). *Mathematical Tables and Other Aids to Computation*, 2(15):97–110, 1946.

[31] M. Harman. The current state and future of search based software engineering. In *Future of Software Engineering*, pages 342–357. IEEE Computer Society, 2007.

[32] David A. Hart. Toward greater artistic control for interactive evolution of images and animation. In Mario Giacobini, editor, *Applications of Evolutionary Computing*, volume 4448 of *LNCS*, pages 527–536. Springer, 2007. ISBN 978-3-540-71804-8.

[33] Ami Hauptman and Moshe Sipper. GP-endchess: Using genetic programming to evolve chess endgame players. In Maarten Keijzer, Andrea Tettamanzi, Pierre Collet, Jano I. van Hemert, and Marco Tomassini, editors, *Proceedings of the 8th European Conference on Genetic Programming*, volume 3447 of *Lecture Notes in Computer Science*, pages 120–131, Lausanne, Switzerland, 30 March - 1 April 2005. Springer. ISBN 3-540-25436-6. doi: doi:10.1007/b107383. URL http://www.cs.bgu.ac.il/~sipper/papabs/eurogpchess-final.pdf.

[34] Thomas Haynes and Sandip Sen. Evolving behavioral strategies in predators and prey. In Gerhard Weiß and Sandip Sen, editors, *Adaptation and Learning in Multiagent Systems*, Lecture Notes in Artificial Intelligence. Springer Verlag, Berlin, Germany, 1995.

[35] Torsten Hildebrandt, Jens Heger, and Bernd Scholz-Reiter. Towards improved dispatching rules for complex shop floor scenarios: a genetic programming approach. In Juergen Branke, Martin Pelikan, Enrique Alba, Dirk V. Arnold, Josh Bongard, Anthony Brabazon, Juergen Branke, Martin V. Butz, Jeff Clune, Myra Cohen, Kalyanmoy Deb, Andries P Engelbrecht, Natalio Krasnogor, Julian F. Miller, Michael O'Neill, Kumara Sastry, Dirk Thierens, Jano van Hemert, Leonardo Vanneschi, and Carsten Witt, editors, *GECCO '10: Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 257–264, Portland, Oregon, USA, 7-11 July 2010. ACM. doi: doi:10.1145/1830483.1830530.

[36] Nguyen Xuan Hoai, R. I. (Bob) McKay, and Daryl Essam. Representation and structural difficulty in genetic programming. *IEEE Transactions on Evolutionary Computation*, 10(2):157–166, April 2006. doi: doi:10.1109/TEVC.2006.871252. URL http://sc.snu.ac.kr/courses/2006/fall/pg/aai/GP/nguyen/Structdiff.pdf.

[37] M. Hofmann, A. Hirschberger, E. Kitzelmannn, and U. Schmid. Inductive synthesis of recursive functional programs. *KI 2007: Advances in Artificial Intelligence*, pages 468–472, 2007. URL http://www.springerlink.com/content/p03v78658627t0l2/.

[38] Amy K. Hoover, M. P. Rosario, and Kenneth O. Stanley. Scaffolding for interactively evolving novel drum tracks for existing songs. In *Proceedings of EvoWorkshops*, volume 4974 of *LNCS*, page 412. Springer, 2008.

[39] G.S. Hornby and J.B. Pollack. Evolving L-systems to generate virtual creatures. *Computers & Graphics*, 25(6):1041–1048, 2001.

[40] M. Hutter. A gentle introduction to the universal algorithmic agent {AIXI}. Technical Report IDSIA-01-03, IDSIA, 2003.

[41] Hitoshi Iba, Hugo de Garis, and Taisuke Sato. Genetic programming using a minimum description length principle. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 12, pages 265–284. MIT Press, 1994. URL http://citeseer.ist.psu.edu/327857.html.

[42] Hitoshi Iba, Yoshihiko Hasegawa, and Topon Kumar Paul. *Applied Genetic Programming and Machine Learning*. CRC Complex and Enterprise Systems Engineering. CRC, 2009. ISBN 1439803692.

[43] Maarten Keijzer. Alternatives in subtree caching for genetic programming. In Maarten Keijzer, Una-May O'Reilly, Simon M. Lucas, Ernesto Costa, and Terence Soule, editors, *Genetic Programming 7th European Conference, EuroGP 2004, Proceedings*, volume 3003 of *LNCS*, pages 328–337, Coimbra, Portugal, 5-7 April 2004. Springer-Verlag. ISBN 3-540-21346-5. URL http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=3003&spage=328.

[44] Maarten Keijzer and Vladan Babovic. Dimensionally aware genetic programming. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1069–1076, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann. ISBN 1-55860-611-4. URL http://www.cs.bham.ac.uk/~wbl/biblio/gecco1999/GP-420.ps.

[45] T. Kötzing, F. Neumann, and R. Spöhel. Pac learning and genetic programming. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 2091–2096. ACM, 2011.

[46] T. Kötzing, F. Neumann, A Sutton, and UM. OŔeilly. The max problem revisited: The importance of mutation in genetic programming. In *GECCO*. ACM, 2011.

[47] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992. ISBN 0-262-11170-5.

[48] John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge Massachusetts, May 1994. ISBN 0-262-11189-6.

[49] John R. Koza, editor. *Late Breaking Papers at the Genetic Programming 1996 Conference Stanford University July 28-31, 1996*, Stanford University, CA, USA, 28–31 July 1996. Stanford Bookstore. ISBN 0-18-201031-7. URL http://www.genetic-programming.org/gp96latebreaking.html.

[50] John R. Koza, Forrest H Bennett III, David Andre, Martin A. Keane, and Frank Dunlap. Automated synthesis of analog electrical circuits by means of genetic programming. *IEEE Transactions on Evolutionary Computation*, 1(2):109–128, July 1997. ISSN 1089-778X. URL http://www.genetic-programming.com/jkpdf/ieeetecjournal1997.pdf.

[51] John R. Koza, David Andre, Forrest H Bennett III, and Martin Keane. *Genetic Programming 3: Darwinian Invention and Problem Solving*. Morgan Kaufman, April 1999. ISBN 1-55860-543-6. URL http://www.genetic-programming.org/gpbook3toc.html.

[52] John R. Koza, Martin A. Keane, Matthew J. Streeter, William Mydlowec, Jessen Yu, and Guido Lanza. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers, 2003. ISBN 1-4020-7446-8. URL http://www.genetic-programming.org/gpbook4toc.html.

[53] W. B. Langdon. How many good programs are there? How long are they? In Kenneth A. De Jong, Riccardo Poli, and Jonathan E. Rowe, editors, *Foundations of Genetic Algorithms VII*, pages 183–202, Torremolinos, Spain, 4-6 September 2002. Morgan Kaufmann. ISBN 0-12-208155-2. URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/wbl_foga2002.pdf. Published 2003.

[54] W. B. Langdon. Large scale bioinformatics data mining with parallel genetic programming on graphics processing units. In Francisco Fernandez de Vega and Erick Cantu-Paz, editors, *Parallel and Distributed Computational Intelligence*, volume 269 of *Studies in Computational Intelligence*, chapter 5, pages 113–141. Springer, January 2010. doi: doi:10.1007/978-3-642-10675-0_6. URL http://www.springer.com/engineering/book/978-3-642-10674-3.

[55] W. B. Langdon and R. Poli. Fitness causes bloat. In P. K. Chawdhry, R. Roy, and R. K. Pant, editors, *Soft Computing in Engineering Design and Manufacturing*, pages 13–22. Springer-Verlag London, 23-27 June 1997. ISBN 3-540-76214-0. URL http://www.cs.bham.ac.uk/~wbl/ftp/papers/WBL.bloat_wsc2.ps.gz.

[56] W. B. Langdon and Riccardo Poli. *Foundations of Genetic Programming*. Springer-Verlag, 2002. ISBN 3-540-42451-2. URL http://www.cs.ucl.ac.uk/staff/W.Langdon/FOGP/.

[57] William B. Langdon. *Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming!*, volume 1 of *Genetic Programming*. Kluwer, Boston, 1998. ISBN 0-7923-8135-1. URL http://www.cs.ucl.ac.uk/staff/W.Langdon/gpdata.

[58] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A generic method for automated software repair. *IEEE Transactions on Software Engineering*, 2011.

[59] Matthew Lewis. Evolutionary visual art and design. In Juan Romero and Penousal Machado, editors, *The art of artificial evolution: a handbook on evolutionary art and music*, pages 3–37. Springer, 2008.

[60] J. Lohn, G. Hornby, and D. Linden. An evolved antenna for deployment on NASAs Space Technology 5 mission. *Genetic Programming Theory and Practice II*, pages 301–315, 2005.

[61] Moshe Looks, Ben Goertzel, and Cassio Pennachin. Learning computer programs with the bayesian optimization algorithm. In Hans-Georg Beyer, Una-May O'Reilly, Dirk V. Arnold, Wolfgang Banzhaf, Christian Blum, Eric W. Bonabeau, Erick Cantu-Paz, Dipankar Dasgupta, Kalyanmoy Deb, James A.

Foster, Edwin D. de Jong, Hod Lipson, Xavier Llora, Spiros Mancoridis, Martin Pelikan, Guenther R. Raidl, Terence Soule, Andy M. Tyrrell, Jean-Paul Watson, and Eckart Zitzler, editors, *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, volume 1, pages 747–748, Washington DC, USA, 25-29 June 2005. ACM Press. ISBN 1-59593-010-8. URL http://www.cs.bham.ac.uk/~wbl/biblio/gecco2005/docs/p747.pdf.

[62] Sean Luke. *Essentials of Metaheuristics*. lulu.com, first edition, 2009. URL http://cs.gmu.edu/~sean/books/metaheuristics/. Available at http://cs.gmu.edu/∼sean/books/metaheuristics/.

[63] Sean Luke. *The ECJ Owner's Manual – A User Manual for the ECJ Evolutionary Computation Library*, zeroth edition, online version 0.2 edition, October 2010. URL http://www.cs.gmu.edu/~eclab/projects/ecj/docs/manual/manual.pdf.

[64] Sean Luke and Liviu Panait. A comparison of bloat control methods for genetic programming. *Evolutionary Computation*, 14(3):309–344, Fall 2006. ISSN 1063-6560. doi: doi:10.1162/evco.2006.14.3.309.

[65] J McCarthy. Programs with common sense. Technical report, Stanford University Dept. Of Computer Science, 1963.

[66] Jon McCormack. Evolutionary L-systems. In Philip F. Hingston, Luigi C. Barone, Zbigniew Michalewicz, and David B. Fogel, editors, *Design by Evolution: Advances in Evolutionary Design*, pages 169–196. Springer-Verlag, 2008.

[67] James McDermott and Una-May O'Reilly. An executable graph representation for evolutionary generative music. In *GECCO '11*, Dublin, 2011.

[68] James McDermott, Jonathan Byrne, John Mark Swafford, Martin Hemberg, Ciaran McNally, Elizabeth Shotton, Erik Hemberg, Michael Fenton, and Michael O'Neill. A comparison of shape grammars and string-rewriting grammars in evolutionary architectural design. *Environment and Planning B*. In press.

[69] James McDermott, Jonathan Byrne, John Mark Swafford, Michael O'Neill, and Anthony Brabazon. Higher-order functions in aesthetic EC encodings. In *2010 IEEE World Congress on Computational Intelligence*, pages 2816–2823, Barcelona, Spain, 18-23 July 2010. IEEE Computation Intelligence Society, IEEE Press. doi: doi:10.1109/CEC.2010.5586077.

[70] R.I. Mckay, N.X. Hoai, P.A. Whigham, Y. Shan, and M. O'Neill. Grammar-based Genetic Programming: a survey. *Genetic Programming and Evolvable Machines*, 11(3):365–396, 2010. ISSN 1389-2576.

[71] Robert I. McKay, Nguyen Xuan Hoai, Peter Alexander Whigham, Yin Shan, and Michael O'Neill. Grammar-based genetic programming: a survey. *Genetic Programming and Evolvable Machines*, 11 (3/4):365–396, September 2010. ISSN 1389-2576. doi: doi:10.1007/s10710-010-9109-y. Tenth Anniversary Issue: Progress in Genetic Programming and Evolvable Machines.

[72] Nicholas Freitag McPhee, Brian Ohs, and Tyler Hutchison. Semantic building blocks in genetic programming. In Michael O'Neill, Leonardo Vanneschi, Steven Gustafson, Anna Isabel Esparcia Alcazar, Ivanoe De Falco, Antonio Della Cioppa, and Ernesto Tarantino, editors, *Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008*, volume 4971 of *Lecture Notes in Computer Science*, pages 134–145, Naples, 26-28 March 2008. Springer. doi: doi:10.1007/978-3-540-78671-9_12.

[73] Julian Miller. What bloat? cartesian genetic programming on boolean problems. In Erik D. Goodman, editor, *2001 Genetic and Evolutionary Computation Conference Late Breaking Papers*, pages 295–302, San Francisco, California, USA, 9-11 July 2001. URL http://www.elec.york.ac.uk/intsys/users/jfm7/gecco2001Late.pdf.

[74] Julian F. Miller and Peter Thomson. Cartesian genetic programming. In Riccardo Poli, Wolfgang Banzhaf, William B. Langdon, Julian F. Miller, Peter Nordin, and Terence C. Fogarty, editors, *Genetic*

Programming, Proceedings of EuroGP'2000, volume 1802 of *LNCS*, pages 121–132, Edinburgh, 15-16 April 2000. Springer-Verlag. ISBN 3-540-67339-3. URL http://www.elec.york.ac.uk/intsys/users/jfm7/cgp-eurogp2000.pdf.

[75] David J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995. doi: doi:10.1162/evco.1995.3.2.199. URL http://vishnu.bbn.com/papers/stgp.pdf.

[76] Alberto Moraglio, Cecilia Di Chio, and Riccardo Poli. Geometric particle swarm optimization. In Marc Ebner, Michael O'Neill, Anikó Ekárt, Leonardo Vanneschi, and Anna Isabel Esparcia-Alcázar, editors, *Proceedings of the 10th European Conference on Genetic Programming*, volume 4445 of *Lecture Notes in Computer Science*, pages 125–136, Valencia, Spain, 11-13 April 2007. Springer. ISBN 3-540-71602-5. doi: doi:10.1007/978-3-540-71605-1_12.

[77] J. Murphy, M. O'Neill, and H. Carr. Exploring grammatical evolution for horse gait optimisation. *Genetic Programming*, pages 183–194, 2009.

[78] Frank Neumann. Computational complexity analysis of multi-objective genetic programming. In *GECCO*. ACM, 2012. to appear.

[79] Peter Nordin. A compiling genetic programming system that directly manipulates the machine code. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 14, pages 311–331. MIT Press, 1994. URL http://cognet.mit.edu/library/books/view?isbn=0262111888.

[80] Peter Nordin, Wolfgang Banzhaf, and Frank D. Francone. Efficient evolution of machine code for CISC architectures using instruction blocks and homologous crossover. In Lee Spector, William B. Langdon, Una-May O'Reilly, and Peter J. Angeline, editors, *Advances in Genetic Programming 3*, chapter 12, pages 275–299. MIT Press, Cambridge, MA, USA, June 1999. ISBN 0-262-19423-6. URL http://www.aimlearning.com/aigp31.pdf.

[81] Mark O'Keeffe and Mel Ó Cinnéide. Search-based refactoring for software maintenance. *Journal of Systems and Software*, 81(4):502–516, 2008.

[82] Michael O'Neill and Anthony Brabazon. Grammatical differential evolution. In Hamid R. Arabnia, editor, *Proceedings of the 2006 International Conference on Artificial Intelligence, ICAI 2006*, volume 1, pages 231–236, Las Vegas, Nevada, USA, June 26-29 2006. CSREA Press. ISBN 1-932415-96-3. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.91.3012.

[83] Michael O'Neill and Conor Ryan. *Grammatical Evolution: Evolutionary Automatic Programming in a Arbitrary Language*, volume 4 of *Genetic programming*. Kluwer Academic Publishers, 2003. ISBN 1-4020-7444-1. URL http://www.wkap.nl/prod/b/1-4020-7444-1.

[84] Michael O'Neill, Erik Hemberg, Conor Gilligan, Eliott Bartley, James McDermott, and Anthony Brabazon. GEVA: Grammatical evolution in java. *SIGEVOlution*, 3(2), Summer 2008. URL http://www.sigevolution.org/issues/pdf/SIGEVOlution200802.pdf.

[85] Michael O'Neill, Leonardo Vanneschi, Steven Gustafson, and Wolfgang Banzhaf. Open issues in genetic programming. *Genetic Programming and Evolvable Machines*, 11(3/4):339–363, September 2010. ISSN 1389-2576. doi: doi:10.1007/s10710-010-9113-2. Tenth Anniversary Issue: Progress in Genetic Programming and Evolvable Machines.

[86] Una-May O'Reilly and Martin Hemberg. Integrating generative growth and evolutionary computation for form exploration. *Genetic Programming and Evolvable Machines*, 8(2):163–186, June 2007. ISSN 1389-2576. doi: doi:10.1007/s10710-007-9025-y. Special issue on developmental systems.

[87] Una-May O'Reilly and Franz Oppacher. Program search with a hierarchical variable length representation: Genetic programming, simulated annealing and hill climbing. In Yuval Davidor, Hans-Paul

Schwefel, and Reinhard Manner, editors, *Parallel Problem Solving from Nature – PPSN III*, number 866 in Lecture Notes in Computer Science, pages 397–406, Jerusalem, 9-14 October 1994. Springer-Verlag. ISBN 3-540-58484-6. URL http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/ppsn-94.ps.gz.

[88] Una-May O'Reilly, Tina Yu, Rick L. Riolo, and Bill Worzel, editors. *Genetic Programming Theory and Practice II*, volume 8 of *Genetic Programming*, Ann Arbor, MI, USA, 13-15 May 2004. Springer. ISBN 0-387-23253-2. URL http://www.springeronline.com/sgw/cda/frontpage/0,11855,5-40356-22-34954683-0,00.html.

[89] M. Orlov and M. Sipper. Flight of the FINCH through the Java wilderness. *Evolutionary Computation, IEEE Transactions on*, 15(2):166–182, 2011. doi: 10.1109/TEVC.2010.2052622.

[90] Tim Perkis. Stack-based genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 148–153, Orlando, Florida, USA, 27-29 June 1994. IEEE Press. URL http://citeseer.ist.psu.edu/432690.html.

[91] Riccardo Poli. Parallel distributed genetic programming. In David Corne, Marco Dorigo, and Fred Glover, editors, *New Ideas in Optimization*, Advanced Topics in Computer Science, chapter 27, pages 403–431. McGraw-Hill, Maidenhead, Berkshire, England, 1999. ISBN 0-07-709506-5. URL http://citeseer.ist.psu.edu/328504.html.

[92] Riccardo Poli and W. B. Langdon. A new schema theory for genetic programming with one-point crossover and point mutation. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 278–285, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann. URL http://citeseer.ist.psu.edu/327495.html.

[93] Riccardo Poli and William B. Langdon. Running genetic programming backward. In Tina Yu, Rick L. Riolo, and Bill Worzel, editors, *Genetic Programming Theory and Practice III*, volume 9 of *Genetic Programming*, chapter 9, pages 125–140. Springer, Ann Arbor, 12-14 May 2005. ISBN 0-387-28110-X. URL http://www.cs.essex.ac.uk/staff/poli/papers/GPTP2005.pdf.

[94] Riccardo Poli and Nicholas Freitag McPhee. General schema theory for genetic programming with subtree-swapping crossover: Part I. *Evolutionary Computation*, 11(1):53–66, March 2003. doi: doi:10.1162/106365603321829005. URL http://cswww.essex.ac.uk/staff/rpoli/papers/ecj2003partI.pdf.

[95] Riccardo Poli and Nicholas Freitag McPhee. General schema theory for genetic programming with subtree-swapping crossover: Part II. *Evolutionary Computation*, 11(2):169–206, June 2003. doi: doi:10.1162/106365603766646825. URL http://cswww.essex.ac.uk/staff/rpoli/papers/ecj2003partII.pdf.

[96] Riccardo Poli and Nicholas Freitag McPhee. A linear estimation-of-distribution GP system. In Michael O'Neill, Leonardo Vanneschi, Steven Gustafson, Anna Isabel Esparcia Alcazar, Ivanoe De Falco, Antonio Della Cioppa, and Ernesto Tarantino, editors, *Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008*, volume 4971 of *Lecture Notes in Computer Science*, pages 206–217, Naples, 26-28 March 2008. Springer. doi: doi:10.1007/978-3-540-78671-9_18.

[97] Riccardo Poli, Jonathan Page, and W. B. Langdon. Smooth uniform crossover, sub-machine code GP and demes: A recipe for solving high-order boolean parity problems. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1162–1169, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann. ISBN 1-55860-611-4. URL http://www.cs.bham.ac.uk/~wbl/biblio/gecco1999/GP-466.pdf.

[98] Riccardo Poli, Nicholas Freitag McPhee, and Jonathan E. Rowe. Exact schema theory and markov chain models for genetic programming and variable-length genetic algorithms with homologous crossover. *Genetic Programming and Evolvable Machines*, 5(1):31–70, March 2004. ISSN 1389-2576. doi: doi:10.1023/B:GENP.0000017010.41337.a7. URL http://cswww.essex.ac.uk/staff/rpoli/papers/GPEM2004.pdf.

[99] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008. URL http://www.gp-field-guide.org.uk. (With contributions by J. R. Koza).

[100] Riccardo Poli, Mario Graff, and Nicholas Freitag McPhee. Free lunches for function and program induction. In *FOGA '09: Proceedings of the tenth ACM SIGEVO workshop on Foundations of genetic algorithms*, pages 183–194, Orlando, Florida, USA, 9-11 January 2009. ACM. ISBN 3-540-27237-2. doi: doi:10.1145/1527125.1527148.

[101] Riccardo Poli, Mathew Salvaris, and Caterina Cinel. Evolution of a brain-computer interface mouse via genetic programming. In Sara Silva, James A. Foster, Miguel Nicolau, Mario Giacobini, and Penousal Machado, editors, *Proceedings of the 14th European Conference on Genetic Programming, EuroGP 2011*, volume 6621 of *LNCS*, pages 203–214, Turin, Italy, 27-29 April 2011. Springer Verlag.

[102] Riccardo Poli, Mathew Salvaris, and Caterina Cinel. Evolution of an effective brain-computer interface mouse via genetic programming with adaptive Tarpeian bloat control. In Rick Riolo, Katya Vladislavleva, and Jason Moore, editors, *Genetic Programming Theory and Practice IX*. Kluwer, 2011.

[103] P. Prusinkiewicz and A. Lindenmayer. *The algorithmic beauty of plants (The Virtual Laboratory)*. Springer, 1991.

[104] Alain Ratle and Michele Sebag. Grammar-guided genetic programming and dimensional consistency: application to non-parametric identification in mechanics. *Applied Soft Computing*, 1(1):105–118, 2001. doi: doi:10.1016/S1568-4946(01)00009-6. URL http://www.sciencedirect.com/science/article/B6W86-43S6W98-B/1/38e0fa6ac503a5ef310e2287be01eff8.

[105] Thomas S. Ray. An approach to the synthesis of life. In C.G Langton, C. Taylor, J.D. Farmer, and S. Rasmussen, editors, *Artificial life II*, pages 371–408. Santa Fe Institute, 1992.

[106] C. Rich and R.C. Waters. Automatic programming: Myths and prospects. *Computer*, 21(8):40–51, 1988.

[107] Justinian P. Rosca. Analysis of complexity drift in genetic programming. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 286–294, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann. URL ftp://ftp.cs.rochester.edu/pub/u/rosca/gp/97.gp.ps.gz.

[108] Conor Ryan. *Automatic Re-engineering of Software Using Genetic Programming*, volume 2 of *Genetic Programming*. Kluwer Academic Publishers, 1 November 1999. ISBN 0-7923-8653-1. URL http://www.wkap.nl/book.htm/0-7923-8653-1.

[109] A.L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 3(3):210, 1959.

[110] Massimiliano Schillaci and Edgar Ernesto Sanchez Sanchez. A brief survey of uGP. *SIGEvolution*, 1 (2):17–21, June 2006.

[111] Jurgen Schmidhuber. Evolutionary principles in self-referential learning. on learning now to learn: The meta-meta-meta...-hook. Diploma thesis, Technische Universitat Munchen, Germany, 14 May 1987. URL http://www.idsia.ch/~juergen/diploma.html.

[112] E. Schulte, S. Forrest, and W. Weimer. Automated program repair through the evolution of assembly code. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 313–316. ACM, 2010.

[113] C. Schumacher, M.D. Vose, and L.D. Whitley. The no free lunch and problem description length. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 565–570, 2001.

[114] Yin Shan, Robert I. McKay, Daryl Essam, and Hussein A. Abbass. A survey of probabilistic model building genetic programming. In M. Pelikan, K. Sastry, and E. Cantu-Paz, editors, *Scalable Optimization via Probabilistic Modeling: From Algorithms to Applications*, volume 33 of *Studies in Computational Intelligence*, chapter 6, pages 121–160. Springer, 2006. ISBN 3-540-34953-7. doi: doi:10.1007/978-3-540-34954-9_6.

[115] Jianhua Shao, James McDermott, Michael O'Neill, and Anthony Brabazon. Jive: a generative, interactive, virtual, evolutionary music system. In Cecilia Di Chio, Anthony Brabazon, Gianni A. Di Caro, Marc Ebner, Muddassar Farooq, Andreas Fink, Jorn Grahl, Gary Greenfield, Penousal Machado, Michael O'Neill, Ernesto Tarantino, and Neil Urquhart, editors, *EvoMUSART*, volume 6025 of *LNCS*, pages 341–350, Istanbul, 7-9 April 2010. Springer. doi: doi:10.1007/978-3-642-12242-2_35.

[116] S. Silva and L. Vanneschi. The importance of being flat—studying the program length distributions of operator equalisation. In Rick Riolo, Katya Vladislavleva, and Jason Moore, editors, *Genetic Programming Theory and Practice IX*, pages 211–233. Springer, 2011.

[117] S. Silva, S. Dignum, and L. Vanneschi. Operator equalisation for bloat free genetic programming and a survey of bloat control methods. *Genetic Programming and Evolvable Machines*, pages 1–42, 2011.

[118] Karl Sims. Artificial evolution for computer graphics. Technical Report TR-185, Thinking Machines Corporation, 1991.

[119] Moshe Sipper. *Evolved to Win*. Lulu, 2011. available at http://www.lulu.com/.

[120] Stephen F. Smith. *A learning system based on genetic adaptive algorithms*. PhD thesis, University of Pittsburgh, 1980.

[121] G. Smits and E. Vladislavleva. Ordinal pareto genetic programming. In Gary G. Yen, Simon M. Lucas, Gary Fogel, Graham Kendall, Ralf Salomon, Byoung-Tak Zhang, Carlos A. Coello Coello, and Thomas Philip Runarsson, editors, *Proceedings of the 2006 IEEE Congress on Evolutionary Computation*, pages 3114–3120, Vancouver, BC, Canada, 16-21 July 2006. IEEE Press. ISBN 0-7803-9487-9. URL http://ieeexplore.ieee.org/servlet/opac?punumber=11108.

[122] Lee Spector. Autoconstructive evolution: Push, pushGP, and pushpop. In Lee Spector, Erik D. Goodman, Annie Wu, W. B. Langdon, Hans-Michael Voigt, Mitsuo Gen, Sandip Sen, Marco Dorigo, Shahram Pezeshk, Max H. Garzon, and Edmund Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 137–146, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann. ISBN 1-55860-774-9. URL http://hampshire.edu/lspector/pubs/ace.pdf.

[123] Lee Spector and Alan Robinson. Genetic programming and autoconstructive evolution with the push programming language. *Genetic Programming and Evolvable Machines*, 3(1):7–40, March 2002. ISSN 1389-2576. doi: doi:10.1023/A:1014538503543. URL http://hampshire.edu/lspector/pubs/push-gpem-final.pdf.

[124] Giovanni Squillero. MicroGP - an evolutionary assembly program generator. *Genetic Programming and Evolvable Machines*, 6(3):247–263, September 2005. ISSN 1389-2576. doi: doi:10.1007/s10710-005-2985-x. Published online: 17 August 2005.

[125] Kenneth O. Stanley. Compositional pattern producing networks: A novel abstraction of development. *Genetic Programming and Evolvable Machines*, 8(2):131–162, June 2007. ISSN 1389-2576. doi: doi: 10.1007/s10710-007-9028-8. Special issue on developmental systems.

[126] M. Suchorzewski and J. Clune. A novel generative encoding for evolving modular, regular and scalable networks. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 1523–1530. ACM, 2011.

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

[128] Julio Tanomaru. Evolving turing machines from examples. In J.-K. Hao, E. Lutton, E. Ronald, M. Schoenauer, and D. Snyers, editors, *Artificial Evolution*, volume 1363 of *LNCS*, Nimes, France, October 1993. Springer-Verlag. URL http://link.springer.de/link/service/series/0558/papers/1363/13630167.pdf.

[129] Stephen Todd and William Latham. *Evolutionary art and computers*. Academic Press, Inc., 1994.

[130] Julian Togelius, Simon Lucas, Ho Duc Thang, Jonathan M. Garibaldi, Tomoharu Nakashima, Chin Hiong Tan, Itamar Elhanany, Shay Berant, Philip Hingston, Robert M. MacCallum, Thomas Haferlach, Aravind Gowrisankar, and Pete Burrow. The 2007 IEEE CEC simulated car racing competition. *Genetic Programming and Evolvable Machines*, 9(4):295–329, December 2008. ISSN 1389-2576. doi: doi:10.1007/s10710-008-9063-0.

[131] Marco Tomassini. *Spatially structured evolutionary algorithms*. Springer, 2005. ISBN 3540241930.

[132] A.M. Turing. Computing machinery and intelligence. *Mind*, 59(236):433–460, 1950.

[133] A.M. Turing. Intelligent machinery. In A. D. J. Evans and Robertson, editors, *Cybernetics: Key Papers*. University Park Press, 1968. Written 1948.

[134] Leonardo Vanneschi, Mauro Castelli, and Sara Silva. Measuring bloat, overfitting and functional complexity in genetic programming. In Juergen Branke, Martin Pelikan, Enrique Alba, Dirk V. Arnold, Josh Bongard, Anthony Brabazon, Juergen Branke, Martin V. Butz, Jeff Clune, Myra Cohen, Kalyanmoy Deb, Andries P Engelbrecht, Natalio Krasnogor, Julian F. Miller, Michael O'Neill, Kumara Sastry, Dirk Thierens, Jano van Hemert, Leonardo Vanneschi, and Carsten Witt, editors, *GECCO '10: Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 877–884, Portland, Oregon, USA, 7-11 July 2010. ACM. doi: doi:10.1145/1830483.1830643.

[135] Ekaterina J. Vladislavleva, Guido F. Smits, and Dick den Hertog. Order of nonlinearity as a complexity measure for models generated by symbolic regression via pareto genetic programming. *IEEE Transactions on Evolutionary Computation*, 13(2):333–349, April 2009. ISSN 1089-778X. doi: doi:10.1109/TEVC.2008.926486.

[136] J. Von Neumann and M.D. Godfrey. First draft of a report on the EDVAC. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.

[137] T. Weise and K. Tang. Evolving distributed algorithms with genetic programming. *Evolutionary Computation, IEEE Transactions on*, 2011. doi: 10.1109/TEVC.2011.2112666. print version forthcoming.

[138] P. A. Whigham. Grammatically-based genetic programming. In Justinian P. Rosca, editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 33–41, Tahoe City, California, USA, 9 July 1995. URL http://divcom.otago.ac.nz/sirc/Peterw/Publications/ml95.zip.

[139] D.R. White, A. Arcuri, and J.A. Clark. Evolutionary improvement of programs. *IEEE Transactions on Evolutionary Computation*, 2011.

[140] Kenneth P. Williams. *Evolutionary algorithms for automatic parallelization*. PhD thesis, University of Reading, 1998.

[141] D.H. Wolpert and W.G. Macready. No free lunch theorems for optimization. *Evolutionary Computation, IEEE Transactions on*, 1(1):67–82, 1997.

[142] John Woodward. Evolving turing complete representations. In Ruhul Sarker, Robert Reynolds, Hussein Abbass, Kay Chen Tan, Bob McKay, Daryl Essam, and Tom Gedeon, editors, *Proceedings of the 2003 Congress on Evolutionary Computation CEC2003*, pages 830–837, Canberra, 8-12 December 2003. IEEE Press. ISBN 0-7803-7804-0. URL http://www.cs.bham.ac.uk/~jrw/publications/2003/EvolvingTuringCompleteRepresentations/cec032e.pdf.

[143] John R. Woodward and James R. Neil. No free lunch, program induction and combinatorial problems. In Conor Ryan, Terence Soule, Maarten Keijzer, Edward Tsang, Riccardo Poli, and Ernesto Costa, editors, *Genetic Programming, Proceedings of EuroGP'2003*, volume 2610 of *LNCS*, pages 475–484, Essex, 14-16 April 2003. Springer-Verlag. ISBN 3-540-00971-X. URL http://www.cs.bham.ac.uk/~jrw/publications/2003/NoFreeLunchProgramInductionandCombinatorialProblems/nfl.ps.

[144] Peter Worth and Susan Stepney. Growing music: musical interpretations of L-systems. In *Evo-MUSART*, pages 545–550. Springer, 2005.

[145] Tina Yu. Hierachical processing for evolving recursive and modular programs using higher order functions and lambda abstractions. *Genetic Programming and Evolvable Machines*, 2(4):345–380, December 2001. ISSN 1389-2576. doi: doi:10.1023/A:1012926821302.

Appendix A

Notes

A.1 Sources

GPEM

Field Guide

This is from Field Guide preface: "As mentioned above, this book started life as a chapter. This was for a forthcoming handbook on computational intelligence edited by John Fulcher and Lakhmi C. Jain. We are grateful to John Fulcher for his useful comments and edits on that book chapter. [Tentatively entitled Computational Intelligence: A Compendium and to be pub- lished by Springer in 2008. http://www.springer.com/engineering/computational+intelligence+and+complexity/book/978-3-540-78292-6]"

Field Guide starts by quoting Turing, Samuel, etc.

Definition: "At the most abstract level GP is a systematic, domain-independent method for getting computers to solve problems automatically starting from a high-level state- ment of what needs to be done." (p. 1).

Selection, crossover, mutation, initialisation, fitness, termination, parameters.

"GP runtime can be estimated by the product of: the number of runs R, the number of generations G, the size of the population P, the average size of the programs s and the number of fitness cases F."—p. 27.

> We think that a system for automatically creating computer programs should create entities that possess most or all of the above essential features of computer programs (or reasonable equivalents thereof). A non-definitional list of attributes for a system for automatically creating computer programs would include the following 16 items:
>
> Attribute No. 1 (Starts with "What needs to be done"): It starts from a high-level statement specifying the requirements of the problem.
>
> Attribute No. 2 (Tells us "How to do it"): It produces a result in the form of a sequence of steps that can be executed on a computer.
>
> Attribute No. 3 (Produces a computer program): It produces an entity that can run on a computer.
>
> Attribute No. 4 (Automatic determination of program size): It has the ability to automatically determine the exact number of steps that must be performed and thus does not require the user to prespecify the size of the solution.
>
> Attribute No. 5 (Code reuse): It has the ability to automatically organize useful groups of steps so that they can be reused.
>
> Attribute No. 6 (Parameterized reuse): It has the ability to reuse groups of steps with different instantiations of values (formal parameters or dummy variables).

Attribute No. 7 (Internal storage): It has the ability to use internal storage in the form of single variables, vectors, matrices, arrays, stacks, queues, lists, relational memory, and other data structures.

Attribute No. 8 (Iterations, loops, and recursions): It has the ability to implement iterations, loops, and recursions.

Attribute No. 9 (Self-organization of hierarchies): It has the ability to automatically organize groups of steps into a hierarchy.

Attribute No. 10 (Automatic determination of program architecture): It has the ability to automatically determine whether to employ subroutines, iterations, loops, recursions, and internal storage, and the number of arguments possessed by each subroutine, iteration, loop, recursion.

Attribute No. 11 (Wide range of programming constructs): It has the ability to implement analogs of the programming constructs that human computer programmers find useful, including macros, libraries, typing, pointers, conditional operations, logical functions, integer functions, floating-point functions, complex-valued functions, multiple inputs, multiple outputs, and machine code instructions.

Attribute No. 12 (Well-defined): It operates in a well-defined way. It unmistakably distinguishes between what the user must provide and what the system delivers.

Attribute No. 13 (Problem-independent): It is problem-independent in the sense that the user does not have to modify the system's executable steps for each new problem.

Attribute No. 14 (Wide applicability): It produces a satisfactory solution to a wide variety of problems from many different fields.

Attribute No. 15 (Scalability): It scales well to larger versions of the same problem.

Attribute No. 16 (Competitive with human-produced results): It produces results that are competitive with those produced by human programmers, engineers, mathematicians, and designers.

—http://www.genetic-programming.com/attributes.html

In his 1983 talk entitled "AI: Where It Has Been and Where It Is Going", machine learning pioneer Arthur Samuel stated the main goal of the fields of machine learning and artificial intelligence: "[T]he aim [is] ... to get machines to exhibit behavior, which if done by humans, would be assumed to involve the use of intelligence."—Koza, http://www.genetic-programming.com/jkpdf/burke2003tutorial.pdf

A.2 Frank's advice

Do you want it to be a survey with lots of references?

Or do you want it to explain the algorithm (kind of like a tutorial), give a bit of history, talk about state of art and open challenges, success stories.

Something like tutorial, history, state of the art, success stories would be great.

Should we cover GP software (commercial and academic)? What about benchmarks? would be good as well (perhaps very brief). What about our research (I would think not, it's more of a textbook, correct?)

perhaps just pointers to what you find currently interesting.

A.3 Length

As a rule of thumb, one (two-column) page in the final layout corresponds to 2.5 one-column pages generated by the Springer LaTeX style. So if you are aiming at 20-30 pages in the handbook, this corresponds to roughly 50-75 one-column LaTeX pages. Alternatively, you can estimate that a page in the final layout contains roughly 5500 characters (including blanks).

James guesses: roughly 24-28 pages with fullpage, excluding TOC and notes.

Proposed budget:
1.5 intro
1.5 history
6 taxonomy
3 applications
6 research topics
1 practicalities
8 citations
27 total