Genetic programming
From Wikipedia, the free encyclopedia

In artificial intelligence, **genetic programming (GP)** is an evolutionary algorithm-based methodology inspired by biological evolution to find computer programs that perform a user-defined task. It is a specialization of genetic algorithms (GA) where each individual is a computer program. It is a machine learning technique used to optimize a population of computer programs according to a fitness landscape determined by a program's ability to perform a given computational task.

<table>
<thead>
<tr>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction</td>
</tr>
<tr>
<td>2 Genetic Programming in a Nutshell</td>
</tr>
<tr>
<td>3 Getting Started</td>
</tr>
<tr>
<td>4 Representation, Initialisation and Operators in Tree-based GP</td>
</tr>
<tr>
<td>4.1 Representation</td>
</tr>
<tr>
<td>4.2 Initialising the Population</td>
</tr>
<tr>
<td>4.3 Selection</td>
</tr>
<tr>
<td>4.4 Recombination and Mutation</td>
</tr>
<tr>
<td>5 Getting Ready to Run Genetic Programming</td>
</tr>
<tr>
<td>5.1 Terminal Set</td>
</tr>
<tr>
<td>5.2 Function Set</td>
</tr>
<tr>
<td>5.2.1 Closure</td>
</tr>
<tr>
<td>5.2.2 Sufficiency</td>
</tr>
<tr>
<td>5.2.3 Evolving Structures other than Programs</td>
</tr>
<tr>
<td>5.3 Fitness Function</td>
</tr>
<tr>
<td>5.4 Genetic Programming Parameters</td>
</tr>
<tr>
<td>5.4.1 Creating the initial GP Population</td>
</tr>
<tr>
<td>5.4.2 Proportion of GP children created by crossover or mutation</td>
</tr>
<tr>
<td>5.4.3 Number of GP children</td>
</tr>
<tr>
<td>5.4.4 Number of Generations, size/depth limits on GP children</td>
</tr>
<tr>
<td>5.4.5 Number and use of Fitness Cases</td>
</tr>
<tr>
<td>5.5 Stopping the Run and Choosing a Solution</td>
</tr>
<tr>
<td>6 Example Genetic Programming Run</td>
</tr>
<tr>
<td>6.1 Preparatory Steps</td>
</tr>
<tr>
<td>6.2 Step-by-Step Sample Run</td>
</tr>
<tr>
<td>6.2.1 Initialisation</td>
</tr>
<tr>
<td>6.2.2 Fitness Evaluation</td>
</tr>
<tr>
<td>6.2.3 Selection, Crossover and Mutation</td>
</tr>
<tr>
<td>6.2.4 Termination and Solution Designation</td>
</tr>
<tr>
<td>7 History</td>
</tr>
<tr>
<td>8 Chromosome representation</td>
</tr>
<tr>
<td>9 Genetic operators</td>
</tr>
<tr>
<td>9.1 Crossover</td>
</tr>
<tr>
<td>9.2 Mutation</td>
</tr>
<tr>
<td>10 Meta-Generic Programming</td>
</tr>
<tr>
<td>11 Implementations</td>
</tr>
</tbody>
</table>
Introduction

The goal of having computers automatically solve problems is central to artificial intelligence (AI) machine learning (ML), and the broad area encompassed by what Turing called "machine intelligence" (Turing, 1948). Machine learning pioneer Arthur Samuel, in his 1983 talk entitled "AI: Where It Has Been and Where It Is Going" (Samuel, 1983), stated that the main goal of the fields of machine learning and artificial intelligence is:

_to get machines to exhibit behaviour, which if done by humans, would be assumed to involve the use of intelligence._

Genetic programming (GP) is an evolutionary computation (EC) technique that automatically solves problems without requiring the user to know or specify the form or structure of the solution in advance. At the most abstract level GP is a _systematic, domain-independent_ method for getting computers to solve problems _automatically_ starting from a _high-level statement_ of what needs to be done.

Since its inception, GP has attracted the interest of myriads of people around the globe.

Genetic Programming in a Nutshell

In genetic programming we evolve a population of _computer programs_. That is, generation by generation, GP _stochastically_ transforms populations of programs into new, hopefully better, populations of programs, cf. Figure [1] (http://cswww.essex.ac.uk/staff/poli/gp-field-guide/Chapter1Introduction.html#x1-16r1) . GP, like nature, is a random process, and it can never guarantee results. GP’s essential randomness, however, can lead it to escape traps by which deterministic methods may be captured. Like nature, GP has been very successful at solving novel and unexpected ways of solving problems. (See (Poli, 2008, Chapter 12)[3] for numerous examples (http://cswww.essex.ac.uk/staff/poli/gp-field-guide/121WhereGPHasDoneWell23.html) .) The basic steps in a GP system are shown in Algorithm 1.1. GP finds out how well a program works by running it, and then comparing its behaviour to some ideal (line 3). We might be interested, for example, in how well a program predicts a time series or controls an industrial process. This comparison is quantified to give a numeric value called _fitness_. Those programs that do well are chosen to breed (line 4) and produce new programs for the next generation (line 5). The primary genetic operators that are used to create new programs from existing ones are:

- **Crossover:** The creation of a child program by combining randomly chosen parts from two selected parent programs.
- **Mutation:** The modification of a child program by altering a randomly chosen part of it.

**Algorithm 1.1: Genetic Programming**

1. Randomly create an initial population of programs from the available primitives (more on this in Section Initialising the Population).
2. repeat
3. _Execute_ each program and ascertain its fitness.
4. _Select_ one or two program(s) from the population with a probability based on fitness to participate in genetic operations (Section on Selection).
5. Create new individual program(s) by applying genetic operations with specified probabilities (Genetic operators).
6. **until** an acceptable solution is found or some other stopping condition is met (e.g., a maximum number of generations is reached).
7. **return** the best-so-far individual.

**Getting Started**

Two key questions for those first exploring GP are:

1. What should I read to get started in GP?
2. Should I implement my own GP system or should I use an existing package? If so, what package should I use?

There are many books on GP, including Koza's four massive contributions. The Field Guide to Genetic Programming[4] is more recent and available online. No single article could claim to be completely comprehensive. The sections at the end of this article (esp. Bibliography) list many sources. There are also a whole host of books, videos, journals, conferences, and on-line sources (including several freely available GP systems) that should be of assistance.

We strongly encourage *doing* GP as well as reading about it; the dynamics of evolutionary algorithms are complex, and the experience of tracing through runs is invaluable. Section Implementations lists many free GP systems. Typically they are distributed as source code.

**Representation, Initialisation and Operators in Tree-based GP**

**Representation**

Traditionally, programs in GP are expressed as syntax tree rather than as lines of code (see linear genetic programming for a simpler alternative). The figure shows the tree representation of the program \[2.2 - \frac{x}{11} + 7 \cos(y)\]. The variables and constants in the program (x, y and 2.2, 11 and 7) are leaves of the tree. In GP they are called *terminals*, whilst the arithmetic operations (+, - *, / and \(\pi\)) are internal nodes called *functions*. The sets of allowed functions and terminals together form the primitive set of a GP system.

```
   +
  /   /
-  *  
/ \
2.2 / \
/ \
X / \
/ \
11 / \\n/ \\
x / \\
11 \\
/ \\
y
```

\[\left(2.2 - \left(\frac{x}{11}\right)\right) + \left(7 \times \cos(y)\right)\]

In more advanced forms of GP, programs can be composed of multiple components (e.g., subroutines). In this case
the representation used in GP is a set of trees (one for each component) grouped together under a special root node that acts as glue, as illustrated in Figure [2] (http://cswww.essex.ac.uk/staff/poli/gp-field-guide/21Representation.html#x1-60r2). We will call these (sub)trees branches. The number and type of the branches in a program, together with certain other features of their structure, form the architecture of the program.

It is common in the GP literature to represent expressions in a prefix notation similar to that used in Lisp or Scheme. For example, 2.2 -x/11 + 7*cos(y) becomes (+ (- 2.2 (/ x 11) (* 7 (cos y))). This notation often makes it easier to see the relationship between (sub)expressions and their corresponding (sub)trees. Therefore we will use trees and their corresponding prefix-notation expressions interchangeably.

How one implements tree-based representation GP trees will obviously depend on the programming languages and libraries being used. Languages that provide automatic garbage collection and dynamic lists as fundamental data types make it easier to implement expression trees and the necessary GP operations. Most traditional languages used in AI research (e.g., Lisp and Prolog), many recent languages (e.g., Ruby and Python), and the languages associated with several scientific programming tools (e.g., MATLAB and Mathematica) have these facilities. In other languages, one may have to implement lists/trees or use libraries that provide such data structures.

In high performance environments, the tree-based representation of programs may be too inefficient since it requires the storage and management of numerous pointers. In some cases, it may be desirable to use GP primitives which accept a variable number of arguments (a quantity we will call arity). An example is the sequencing instruction progn, which accepts any number of arguments, executes them one at a time and then returns the value returned by the last argument. However, fortunately, it is now extremely common in GP applications for all functions to have a fixed number of arguments. If this is the case, then, the brackets in prefix-notation expressions are redundant, and trees can efficiently be represented as simple linear sequences. In effect, the function's name gives its arity and from the arities the brackets can be inferred. For example, the expression (+ (- 2.2 (/ x 11) (* 7 (cos y)))) could be written unambiguously as the sequence + - 2.2 / x 11 * 7 cos y.

The choice of whether to use such a linear representation or an explicit tree representation is typically guided by questions of convenience, efficiency, the genetic operations being used (some may be more easily or more efficiently implemented in one representation), and other data one may wish to collect during runs. (It is sometimes useful to attach additional information to nodes, which may be easier to implement if they are explicitly represented).

These tree representations are the most common in GP, e.g., numerous high-quality, freely available GP implementations use them (see implementations). However, there are other important representations, e.g. linear and graph based GP.

**Initialising the Population**

Like in other evolutionary algorithms, in GP the individuals in the initial population are typically randomly generated. There are a number of different approaches to generating this random initial population. Here we will describe two of the simplest (and earliest) methods (the full and grow methods), and a widely used combination of the two known as Ramped half-and-half.

In both the full and grow methods, the initial individuals are generated so that they do not exceed a user specified maximum depth. The depth of a node is the number of edges that need to be traversed to reach the node starting from the tree's root node (which is assumed to be at depth 0). The depth of a tree is the depth of its deepest leaf (e.g., the tree in first figure has a depth of 3). In the full method (so named because it generates full trees, i.e. all leaves are at the same depth) nodes are taken at random from the function set until the maximum tree depth is reached. (Beyond that depth, only terminals can be chosen.) Figure [3] (http://cswww.essex.ac.uk/staff/poli/gp-field-guide/22InitialisingThePopulation.html#dx1-79) shows a series of snapshots of the construction of a full tree of depth 2. The children of the * and / nodes must be leaves or otherwise the tree would be too deep. Thus, at both steps t=3, t=4, t=6 and t=7 a terminal must be chosen. (x, y, 1 and 0 were chosen).
Although, the full method generates trees where all the leaves are at the same depth, this does not necessarily mean that all initial trees will have an identical number of nodes (often referred to as the size of a tree) or the same shape. This only happens, in fact, when all the functions in the primitive set have an equal arity. Nonetheless, even when mixed-arity primitive sets are used, the range of program sizes and shapes produced by the full method may be rather limited. The grow method, on the contrary, allows for the creation of trees of more varied sizes and shapes. Nodes are selected from the whole primitive set (i.e., functions and terminals) until the depth limit is reached. Once the depth limit is reached only terminals may be chosen (just as in the full method). Figure [4] (http://www.essex.ac.uk/staff/poli/gp-field-guide/22InitialisingthePopulation.html#dx1-83) illustrates this process for the construction of a tree with depth limit 2. Here the first argument of the + root node happens to be a terminal. This closes off that branch preventing it from growing any more before it reached the depth limit. The other argument is a function (-), but its arguments are forced to be terminals to ensure that the resulting tree does not exceed the depth limit. Pseudocode for a recursive implementation of both the full and grow methods is given in Algorithm 2.1.

\[
\text{function gen_rnd_expr(func_set, term_set, max_d, method)}
\]

1. if max_d = 0 or (method = grow and random then
2. expr = choose_random_element(term_set)
3. else
4. func = choose_random_element(func_set)
5. for i = 1 to arity(func)
6. arg_i = gen_rnd_expr(func_set, term_set, max_d - 1, method)
7. endfor
8. expr = (func, arg_1, arg_2, ...)
9. endif
10. return expr

Notes: func_set is a function set, term_set is a terminal set, max_d is the maximum allowed depth for expressions, method is either full or grow, expr is the generated expression in prefix notation.

**Algorithm 2.1: Pseudocode for recursive program generation with the full and grow methods.**

Because neither the grow or full method provide a very wide array of sizes or shapes on their own, Koza\[^5\] proposed a combination called *ramped half-and-half*. Half the initial population is constructed using full and half is constructed using grow. This is done using a range of depth limits (hence the term *ramped*) to help ensure that we generate trees having a variety of sizes and shapes.

While these methods are easy to implement and use, they often make it difficult to control the statistical distributions of important properties such as the sizes and shapes of the generated trees. For example, the sizes and shapes of the trees generated via the grow method are highly sensitive to the sizes of the function and terminal sets. If, for example, one has significantly more terminals than functions, the grow method will almost always generate very short trees regardless of the depth limit. Similarly, if the number of functions is considerably greater than the number of terminals, then the grow method will behave quite similarly to the full method. The arities of the functions in the primitive set also influence the size and shape of the trees produced by grow. Page 50 of the Field Guide describes other initialisation mechanisms which address these issues.

The initial population need not be entirely random. If something is known about likely properties of the desired solution, trees having these properties can be used to seed the initial population.

**Selection**

As with most evolutionary algorithms, genetic operators in GP are applied to individuals that are probabilistically selected based on fitness. That is, better individuals are more likely to have more child programs than inferior
individuals. The most commonly employed method for selecting individuals in GP is tournament selection, which is discussed below, followed by fitness-proportionate selection, but any standard evolutionary algorithm selection mechanism can be used.

In tournament selection a number of individuals are chosen at random from the population. These are compared with each other and the best of them is chosen to be the parent. When doing crossover, two parents are needed and, so, two selection tournaments are made. Note that tournament selection only looks at which program is better than another. It does not need to know how much better. This effectively automatically rescales fitness, so that the selection pressure on the population remains constant. Thus, a single extraordinarily good program cannot immediately swamp the next generation with its children; if it did, this would lead to a rapid loss of diversity with potentially disastrous consequences for a run. Conversely, tournament selection amplifies small differences in fitness to prefer the better program even if it is only marginally superior to the other individuals in a tournament.

An element of noise is inherent in tournament selection due to the random selection of candidates for tournaments. So, while preferring the best, tournament selection does ensure that even average-quality programs have some chance of having children. Since tournament selection is easy to implement and provides automatic fitness rescaling, it is commonly used in GP.

Considering that selection has been described many times in the evolutionary algorithms literature, we will not provide details of the numerous other mechanisms that have been proposed. (Goldberg, 1989)[6], for example, describes fitness-proportionate selection, stochastic universal sampling and several others.

**Recombination and Mutation**

GP departs significantly from other evolutionary algorithms in the implementation of the operators of crossover and mutation. The most commonly used form of crossover is **subtree crossover**. Given two parents, subtree crossover randomly (and independently) selects a crossover point (a node) in each parent tree. Then, it creates the offspring by replacing the subtree rooted at the crossover point in a copy of the first parent with a copy of the subtree rooted at the crossover point in the second parent, as illustrated in Figure [5] (http://cswww.essex.ac.uk/staff/poli/gp-field-guide/24RecombinationandMutation.html#dx1-123). Copies are used to avoid disrupting the original individuals. This way, if selected multiple times, they can take part in the creation of multiple offspring programs. Note that it is also possible to define a version of crossover that returns two offspring, but this is not commonly used.

Often crossover points are not selected with uniform probability. Typical GP primitive sets (using binary functions of arity 2) lead to trees with an average branching factor (the number of children of each node) of two, so the majority of the nodes will be leaves. Consequently the uniform selection of crossover points leads to crossover operations frequently exchanging only very small amounts of genetic material (i.e., small subtrees); many crossovers may in fact reduce to simply swapping two leaves. To counter this, Koza[7] suggested the widely used approach of choosing functions 90% of the time and leaves 10% of the time. Many other types of crossover and mutation of GP trees are possible.

The most commonly used form of mutation in GP (which we will call **subtree mutation**) randomly selects a mutation point in a tree and substitutes the subtree rooted there with a randomly generated subtree. This is illustrated in Figure [6] (http://cswww.essex.ac.uk/staff/poli/gp-field-guide/24RecombinationandMutation.html#dx1-125). Subtree mutation is sometimes implemented as crossover between a program and a newly generated random program; this operation is also known as **headless chicken** crossover.[8]

Another common form of mutation is **point mutation**, which is GP's rough equivalent of the bit-flip mutation used in genetic algorithms.[9] In point mutation, a random node is selected and the primitive stored there is replaced with a different random primitive of the same arity taken from the primitive set. If no other primitives with that arity exist, nothing happens to that node (but other nodes may still be mutated). When subtree mutation is applied, this involves the modification of exactly one subtree. Point mutation, on the other hand, is typically applied on a per-node basis. That is, each node is considered in turn and, with a certain probability, it is altered as explained above.
This allows multiple nodes to be mutated independently in one application of point mutation.

The choice of which of the operators described above should be used to create an offspring is probabilistic. Operators in GP are normally mutually exclusive (unlike other evolutionary algorithms where offspring are sometimes obtained via a composition of operators). Their probability of application are called operator rates. Typically, crossover is applied with the highest probability, the crossover rate often being 90 percent or higher. On the contrary, the mutation rate is much smaller, typically being in the region of one percent.

When the rates of crossover and mutation add up to a value p which is less than 100%, an operator called reproduction is also used, with a rate of 1-p. Reproduction simply involves the selection of an individual based on fitness and the insertion of a copy of it in the next generation.

**Getting Ready to Run Genetic Programming**

To apply a GP system to a problem, several decisions need to be made; these are often termed the preparatory steps. The key choices are:

1. What is the terminal set?
2. What is the function set?
3. What is the fitness measure?
4. What parameters will be used for controlling the run?
5. What will be the termination criterion, and what will be designated the result of the run?

**Terminal Set**

While it is common to describe GP as evolving programs, GP is not typically used to evolve programs in the familiar Turing-complete languages humans normally use for software development. It is instead more common to evolve programs (or expressions or formulae) in a more constrained and often domain-specific language. The first two preparatory steps, the definition of the terminal and function sets, specify such a language. That is, together they define the ingredients that are available to GP to create computer programs.

The terminal set may consist of:

- *the program's external inputs*. These typically take the form of named variables (e.g., x, y).
- *functions with no arguments*. These may be included because they return different values each time they are used, such as the function rand() which returns random numbers, or a function dist_to_wall() that returns the distance to an obstacle from a robot that GP is controlling. Another possible reason is because the function produces side effects. Functions with side effects do more than just return a value: they may change some global data structures, print or draw something on the screen, control the motors of a robot.
- *constants*. These can be pre-specified, randomly generated as part of the tree creation process, or created by mutation.

Using a primitive such as rand can cause the behaviour of an individual program to vary every time it is called, even if it is given the same inputs. This is desirable in some applications. However, we more often want a set of fixed random constants that are generated as part of the process of initialising the population. This is typically accomplished by introducing a terminal that represents an ephemeral random constant. Every time this terminal is chosen in the construction of an initial tree (or a new subtree to use in an operation like mutation), a different random value is generated which is then used for that particular terminal, and which will remain fixed for the rest of the run. The use of ephemeral random constants is typically denoted by including the symbol R in the terminal set.

**Function Set**
The function set used in GP is typically driven by the nature of the problem domain. In a simple numeric problem, for example, the function set may consist of merely the arithmetic functions (+, -, *, /). However, all sorts of other functions and constructs typically encountered in computer programs can be used. Sometimes the primitive set includes specialised functions and terminals which are designed to solve problems in a specific problem domain. For example, if the goal is to program a robot to mop the floor, then the function set might include such actions as move, turn, and swish-the-mop.

Closure

For GP to work effectively, most function sets are required to have an important property known as closure, which can in turn be broken down into the properties of type consistency and evaluation safety.

Type consistency is required because subtree crossover can mix and join nodes arbitrarily. As a result it is necessary that any subtree can be used in any of the argument positions for every function in the function set, because it is always possible that subtree crossover will generate that combination. It is thus common to require that all the functions be type consistent, i.e., they all return values of the same type, and that each of their arguments also have this type. For example, +, -, *, and / can be defined so that they each take two integer arguments and return an integer. Sometimes type consistency can be weakened somewhat by providing an automatic conversion mechanism between types. We can, for example, convert numbers to Booleans by treating all negative values as false, and non-negative values as true. However, conversion mechanisms can introduce unexpected biases into the search process, so they should be used with care.

The type consistency requirement can seem quite limiting but often simple restructuring of the functions can resolve apparent problems. For example, an if function is often defined as taking three arguments: the test, the value to return if the test evaluates to true and the value to return if the test evaluates to false. The first of these three arguments is clearly Boolean, which would suggest that it can't be used with numeric functions like +. This, however, can easily be worked around by providing a mechanism to convert a numeric value into a Boolean automatically as discussed above. Alternatively, one can replace the 3-input if with a function of four (numeric) arguments a, b, c, d. The 4-input if implements "If a<b then return value c otherwise return value d".

An alternative to requiring type consistency is to extend the GP system. Crossover and mutation might explicitly make use of type information so that the children they produce do not contain illegal type mismatches. When mutating a legal program, for example, mutation might be required to generate a subtree which returns the same type as the subtree it has just deleted.

The other component of closure is evaluation safety. Evaluation safety is required because many commonly used functions can fail at run time. An evolved expression might, for example, divide by zero, or call MOVE_FORWARD when facing a wall or precipice. This is typically dealt with by modifying the normal behaviour of primitives. It is common to use protected versions of numeric functions that can otherwise throw exceptions, such as division, logarithm, exponential and square root. The protected version of a function first tests for potential problems with its input(s) before executing the corresponding instruction; if a problem is spotted then some default value is returned. Protected division (often noted with %) checks to see if its second argument is 0. If so, % typically returns the value 1 (regardless of the value of the first argument). Similarly, in a robotic application a MOVE_AHEAD instruction can be modified to do nothing if a forward move is illegal or if moving the robot might damage it.

An alternative to protected functions is to trap run-time exceptions and strongly reduce the fitness of programs that generate such errors. However, if the likelihood of generating invalid expressions is very high, this can lead to too many individuals in the population having nearly the same (very poor) fitness. This makes it hard for selection to choose which individuals might make good parents.

One type of run-time error that is more difficult to check for is numeric overflow. If the underlying implementation system throws some sort of exception, then this can be handled either by protection or by penalising as discussed
above. However, it is common for implementation languages to ignore integer overflow quietly and simply wrap around. If this is unacceptable, then the GP implementation must include appropriate checks to catch and handle such overflows.

**Sufficiency**

There is one more property that primitives sets should have: *sufficiency*. Sufficiency means it is possible to express a solution to the problem at hand using the elements of the primitive set. Unfortunately, sufficiency can be guaranteed only for those problems where theory, or experience with other methods, tells us that a solution can be obtained by combining the elements of the primitive set.

As an example of a sufficient primitive set consider \{AND, OR, NOT, x1, x2, ..., xN\}. It is always sufficient for Boolean induction problems, since it can produce all Boolean functions of the variables x1, x2, ..., xN. Where as \{+, -, *, /, x, 0, 1, 2\}, is insufficient, since it is not able to represent transcendental functions. The function exp(x), for example, is transcendental and therefore cannot be expressed as a rational function (basically, a ratio of polynomials), and so cannot be represented exactly by any combination of \{+, -, *, /, x, 0, 1, 2\}. When a primitive set is insufficient, GP can only develop programs that approximate the desired one. However, in many cases such an approximation can be very close and good enough for the user's purpose. Adding a few unnecessary primitives in an attempt to ensure sufficiency does not tend to slow down GP overmuch, although there are cases where it can bias the system in unexpected ways.

**Evolving Structures other than Programs**

There are many problems where solutions cannot be directly cast as computer programs. For example, in many design problems the solution is an artifact of some type: a bridge, a circuit, an antenna, a lens, etc. GP has been applied to problems of this kind by using a trick: the primitive set is set up so that the evolved programs construct solutions to the problem. This is analogous to the process by which an egg grows into a chicken. For example, if the goal is the automatic creation of an electronic controller for a plant, the function set might include common components such as integrator, differentiator, wire, time lag, and gain, and the terminal set might contain reference, signal, and plant output. Each of these primitives, when executed, inserts the corresponding device into the controller being built. If, on the other hand, the goal is to synthesise analogue electrical circuits, the function set might include components such as transistors, capacitors, resistors, etc.

**Fitness Function**

The first two preparatory steps define the primitive set for GP, and therefore indirectly define the search space GP will explore. This includes all the programs that can be constructed by composing the primitives in all possible ways. However, at this stage, we still do not know which elements or regions of this search space are good. I.e., which regions of the search space include programs that solve, or approximately solve, the problem. This is the task of the fitness measure, which is our primary (and often sole) mechanism for giving a high-level statement of the problem’s requirements to the GP system. For example, suppose the goal is to get GP to synthesise an amplifier automatically. Then the fitness function is the mechanism which tells GP to synthesise a circuit that amplifies an incoming signal. (As opposed to evolving a circuit that suppresses the low frequencies of an incoming signal, or computes its square root, etc. etc.)

Fitness can be measured in many ways. For example, in terms of: the amount of error between its output and the desired output; the amount of time (fuel, money, etc.) required to bring a system to a desired target state; the accuracy of the program in recognising patterns or classifying objects; the payoff that a game-playing program produces; the compliance of a structure with user-specified design criteria.

There is something unusual about the fitness functions used in GP that differentiates them from those used in most other evolutionary algorithms. Because the structures being evolved in GP are computer programs, fitness
evaluation normally requires executing all the programs in the population, typically multiple times. While one can compile the GP programs that make up the population, the overhead of building a compiler is usually substantial, so it is much more common to use an interpreter to evaluate the evolved programs.

Interpreting a program tree means executing the nodes in the tree in an order that guarantees that nodes are not executed before the value of their arguments (if any) is known. This is usually done by traversing the tree recursively starting from the root node, and postponing the evaluation of each node until the values of its children (arguments) are known. Other orders, such as going from the leaves to the root, are possible. If none of the primitives have side effects, the two orders are equivalent. This depth-first recursive process is illustrated in Figure 7 (http://cswwww.essex.ac.uk/staff/poli/gp-field-guide/33Step3FitnessFunction.html#fn12-bk) Algorithm 3.1 gives a pseudocode implementation of the interpretation procedure. The code assumes that programs are represented as prefix-notation expressions and that such expressions can be treated as lists of components.

```plaintext
procedure: eval(expr)
if expr is a list then
  proc = expr(1)
  if proc is a function then
    value = proc( eval(expr(2)), eval(expr(3)), ... )  // Function: evaluate arguments
  else
    value = proc(expr(2), expr(3), ...)
  end if
else
  if expr is a variable or expr is a constant then
    value = expr
    return value  // Terminal variable or constant: just read the value
  else
    value = expr()
    return value  // Terminal 0-arity function: execute
  end if
end if
```

Notes: expr is an expression in prefix notation, expr(1) represents the primitive at the root of the expression, expr(2) represents the first argument of that primitive, expr(3) represents the second argument, etc.

**Algorithm 3.1 Interpreter for genetic programming**

In some problems we are interested in the *output* produced by a program, namely the value returned when we evaluate the tree starting at the root node. In other problems we are interested in the actions performed by a program composed of functions with side effects. In either case the fitness of a program typically depends on the results produced by its execution on many different inputs or under a variety of different conditions. For example the program might be tested on all possible combinations of inputs $x_1, x_2, ..., x_N$. Alternatively, a robot control program might be tested with the robot in a number of starting locations. These different test cases typically contribute to the fitness value of a program incrementally, and for this reason are called *fitness cases*.

Another common feature of GP fitness measures is that, for many practical problems, they are *multi-objective*, i.e., they combine two or more different elements that are often in competition with one another. The area of multi-objective optimisation is a complex and active area of research in GP and machine learning in general.¹⁰

**Genetic Programming Parameters**

The fourth preparatory step specifies the control parameters for the run. The most important control parameter is the *population size*. Other control parameters include the *probabilities of performing the genetic operations*, the *maximum size* for programs and other details of the run.

It is impossible to make general recommendations for setting optimal parameter values, as these depend too much
It is impossible to make general recommendations for setting\textit{ optimal} parameter values, as these depend too much on the details of the application. However, genetic programming is in practice robust, and it is likely that many different parameter values will work. As a consequence, one need not typically spend a long time tuning GP for it to work adequately.

\textbf{Creating the initial GP Population}

It is common to create the initial population randomly using ramped half-and-half with a depth range of 2-6. The initial tree sizes will depend upon the number of the functions, the number of terminals and the arities of the functions. However, evolution will quickly move the population away from its initial distribution.

\textbf{Proportion of GP children created by crossover or mutation}

Traditionally, 90 percent of children are created by subtree crossover. However, the use of a 50-50 mixture of crossover and a variety of mutations also appears to work well.

\textbf{Number of GP children}

In many cases, the main limitation on the population size is the time taken to evaluate the fitnesses, not the space required to store the individuals. As a rule one prefers to have the largest population size that your system can handle gracefully; normally, the population size should be at least 500, and people often use much larger populations. Often, to a first approximation, 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 $s$s and the number of fitness cases F.

\textbf{Number of Generations, size/depth limits on GP children}

Typically, the number of generations is limited to between ten and fifty; the most productive search is usually performed in those early generations, and if a solution hasn’t been found then, it's unlikely to be found in a reasonable amount of time. The folk wisdom on population size is to make it as large as possible, but there are those who suggest using many runs with much smaller populations instead. Some implementations do not require arbitrary limits of tree size. Even so, because of bloat (the uncontrolled growth of program sizes during GP runs, it is common to impose either a size or a depth limit or both.

\textbf{Number and use of Fitness Cases}

Sometimes the number of fitness cases is limited by the amount of training data available. In this case, the fitness function should use all of it. One does not necessarily need to use verification or holdout data, since over-fitting can be avoided by other means. In other cases, e.g. 22-bit even parity, there can almost be too much training data. Then the fitness function may be reduced to use just a subset of the training data. This does not necessarily have to be done manually as there are a number of algorithms that dynamically change the test set as the GP runs.

It is common to record the GP parameters in a table.

\textbf{Stopping the Run and Choosing a Solution}

The fifth preparatory step consists of specifying the \textit{termination criterion} and the method of \textit{designating the result} of the run. The termination criterion may include a maximum number of generations to be run as well as a problem-specific success predicate, such as the error falls below one percent. Typically, the best-so-far program is then harvested and designated as the result of the run, although one might wish to return additional individuals and data as necessary or appropriate for the problem domain.
**Example Genetic Programming Run**

This is an illustrative run of GP in which the goal is to automatically create a program with a target input/output behaviour. In particular, we want to evolve an expression whose values match those of \(x^*x+x+1\) in the range \([-1, +1]\). The process of mechanically creating a computer program that fits certain numerical data is sometimes called *system identification* or *symbolic regression*.

We begin with the five preparatory steps from the previous section and then describe in detail the events in one run.

**Preparatory Steps**

The purpose of the first two preparatory steps is to specify the ingredients the evolutionary process can use to construct potential solutions. Because the problem is to find a mathematical function of one independent variable, \(x\), the terminal set (the inputs of the to-be-evolved programs) must include this variable. The terminal set also includes ephemeral random constants drawn from some reasonably range. What is a "reasonable" range is likely to be *extremely* problem dependent. While in theory large constants can be built using small constants and arithmetic operators, the performance of a system is likely to improve considerably if constants of roughly the right magnitude are provided from the beginning. The choice of genetic operators can also be important here. If a system is struggling to evolve effective constants, it may be helpful to introduce mutation operators specifically designed to search of the space of constants. In this example we chose constants randomly from -5.0 to +5.0, as described in Section 3.1. Thus the terminal set, \(T\), is \(T = \{x, \text{Re}\}\).

The statement of the problem does not specify which functions may be employed in the to-be-evolved program. One simple choice for the function set is the four ordinary arithmetic functions: addition, subtraction, multiplication and division. Most numeric regression problems will require at least these operations, sometimes with additional functions such as sin and log. We will use the simple function set \(F = \{+,-,\ast,\div\}\), where \(\div\) is protected division as discussed in Section 3.2.1 The target polynomial can be expressed exactly using the terminal and function sets we have chosen, so these primitives are sufficient for the quadratic polynomial problem. (cf. Sufficiency above.)

The third preparatory step involves constructing the fitness measure that specifies what the user wants. The high-level goal of this problem is to find a program whose output is equal to the values of the quadratic polynomial \(x^*x+x+1\). Therefore, the fitness assigned to a particular individual in the population must reflect how closely the output of an individual program comes to the target polynomial \(x^*x + x + 1\).

In principle, the fitness measure could be defined in terms of the mathematical integral of the difference between the evolved function and the target function. However, for most symbolic regression problems, it is not practical or even possible to compute the value of the integral analytically. Thus, it is common to define the fitness to be the *sum of absolute errors* measured at different values of the independent variable \(x\) in the range \([-1.0, +1.0\). In particular, we will measure the errors for \(x\) \([-1.0, -0.9, ..., 0.9, 1.0\]. A smaller value of fitness (error) is better; a fitness (error) of zero would indicate a perfect fit. With this definition, our fitness is (approximately) proportional to the area between the parabola \(x^*x+x+1\) and the curve representing the candidate individual (see Figure 4.2 (http://dcs.essex.ac.uk/staff/ropoli/gp-field-guide/42StepbyStepSampleRun.html#x1-289r2) for examples).

The fourth step is where we set our run parameters. The population size in this small illustrative example will be just four. The population size for a run of GP typically consists of thousands of individuals, but we will use this tiny population size to keep the example manageable. The crossover operation is commonly used to generate about 90% of the individuals in the population; the reproduction operation (where a fit individual is simply copied from one generation to the next) is used to generate about 8% of the population; the mutation operation is used to generate about 1% of the population; and the architecture-altering operations are used to generate perhaps 1% of the population. However, because this example involves an abnormally small population of only four individuals, the crossover operation will be used twice (each time generating one individual), which corresponds to a crossover rate of 50%, while the mutation and reproduction operations will each be used to generate one individual. These are therefore applied with a rate of 25% each. For simplicity, the architecture-altering operations are not used for this problem.
In the fifth and final step we need to specify a termination condition. A reasonable termination criterion for this problem is that the run will continue from generation to generation until the fitness (or error) of some individual is less than 0.1. In this contrived example, our example run will (atypically) yield an algebraically perfect solution with a fitness of zero after just one generation.

**Step-by-Step Sample Run**

Now that we have performed the five preparatory steps, the run of GP can be launched. The GP setup is summarised in the Table:

<table>
<thead>
<tr>
<th>Parameters for example genetic programming run</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Objective</strong></td>
</tr>
<tr>
<td><strong>Function set</strong></td>
</tr>
<tr>
<td><strong>Terminal set</strong></td>
</tr>
<tr>
<td><strong>Fitness</strong></td>
</tr>
<tr>
<td><strong>Selection</strong></td>
</tr>
<tr>
<td><strong>Initial population</strong></td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
</tr>
<tr>
<td><strong>Termination</strong></td>
</tr>
</tbody>
</table>

**Initialisation**

GP starts by randomly creating a population of four individual computer programs. The four programs are shown in Figure [8] (http://dces.essex.ac.uk/staff/ropoli/gp-field-guide/42StepbyStepSampleRun.html#x1-288r1%7C4.1) in the form of trees.

The first randomly constructed program tree is equivalent to the expression \( x+1 \). The second program adds the constant terminal 1 to the result of multiplying \( x \) by \( x \) and is equivalent to \( x^2 + x + 1 \). The third program adds the constant terminal 2 to the constant terminal 0 and is equivalent to the constant value 2. The fourth program is equivalent to \( x \).

**Fitness Evaluation**

Randomly created computer programs will typically be very poor at solving any problem. However, even in a population of randomly created programs, some programs are better than others. The four random individuals from generation 0 in Figure 4.1 produce outputs that deviate by different amounts from the target function \( x^2 + x + 1 \). Figure 4.2 compares the plots of each of the four individuals in Figure 4.1 and the target quadratic function \( x^2 + x + 1 \). The sum of absolute errors for the straight line \( x+1 \) (the first individual) is 7.7 The sum of absolute errors for the parabola \( x^2 + x + 1 \) (the second individual) is 11.0 and 28.7

As can be seen in Figure 4.2, the straight line \( x+1 \) is closer to the parabola \( x^2 + x + 1 \) in the range from -1 to +1 than any of three other programs in the population. This straight line is, of course, not equivalent to the parabola \( x^2 + x + 1 \); it is not even a quadratic function. It is merely the best candidate that happened to emerge from the blind (and very limited) random search of generation 0.

*In the valley of the blind, the one-eyed man is king.*
After the fitness of each individual in the population is found, GP then probabilistically selects the fitter programs from the population to act as the parents of the next generation. The genetic operations are applied to the selected individuals to create offspring programs. The important point is that our selection process is not greedy. Individuals that are known to be inferior still have some chance of being selected. The best individual in the population is not guaranteed to be selected and the worst individual in the population will not necessarily be excluded.

In this example we will start with the reproduction operation. Because the first individual is the most fit individual in the population, it is very likely to be selected to participate in a genetic operation. Let us suppose that this particular individual is, in fact, selected for reproduction. If so, it is copied, without alteration, into the next generation (generation 1). It is shown in Figure 4.2a (http://dces.essex.ac.uk/staff/rpoli/gp-field-guide/42StepbyStepSampleRun.html#x1-294r3) as part of the population of the new generation.

We next perform the mutation operation. Because selection is probabilistic, it is possible that the third best individual in the population is selected. One of the three nodes of this individual is then randomly picked as the site for the mutation. In this example, the constant terminal 2 is picked as the mutation site. This program is then randomly mutated by deleting the entire subtree rooted at the picked point (in this case, just the constant terminal 2) and inserting a subtree that is randomly constructed in the same way that the individuals of the initial random population were originally created. In this particular instance, the randomly grown subtree computes $x$ divided by $x$ using the protected division operation % . The resulting individual is shown in Figure 4.2b. This particular mutation changes the original individual from one having a constant value of 2 into one having a constant value of 1, improving its fitness from 17.98 to 11.0.

Finally, we use the crossover operation to generate our final two individuals for the next generation. Because the first and second individuals in generation 0 are both relatively fit, they are likely to be selected to participate in crossover. However, selection can always pick suboptimal individuals. So, let us assume that in our first application of crossover the pair of selected parents is composed of the above-average tree in Figure 4.1a and the below-average tree in Figure 4.1d. One point of the first parent, namely the + function is randomly picked as the crossover point for the first parent. One point of the second parent, namely the leftmost terminal $x$ is randomly picked as the crossover point for the second parent. The crossover operation is then performed on the two parents. The offspring (Figure 4.3c) is equivalent to $x^*x+x+1$ and is not particularly noteworthy.

Let us now assume, that in our second application of crossover, selection chooses the two most fit individuals as parents: the individual in Figure 4.1b as the first parent, and the individual in Figure 4.1a as the second. Let us further imagine that crossover picks the leftmost terminal $x$ in Figure 4.1b as a crossover point for the first parent, and the + function in Figure 4.1a as the crossover point for the second parent. Now the offspring (Figure 4.3d) is equivalent to $x^*x+x+1$ and has a fitness (sum of absolute errors) of zero.

**Termination and Solution Designation**

Because the fitness of the individual in Figure 4.3d is below 0.1, the termination criterion for the run is satisfied and the run is automatically terminated. This best-so-far individual (Figure 4.3d) is the result of the GP run.

Note that the best-of-run individual incorporates a good trait (the quadratic term $x^*x$) from the first parent with two other good traits (the linear term $x$ and constant term of 1) from the second parent. The crossover operation thus produced a solution to this problem by recombining good traits from these two relatively fit parents into a superior (indeed, perfect) offspring.

This is, obviously, a highly simplified example, and the dynamics of a real GP run are typically far more complex than we have presented. Also, in general, there is no guarantee that an exact solution like this will be found by GP.

**History**

In 1954, GP began with the evolutionary algorithms first used by Nils Aall Barricelli applied to evolutionary
In 1974, G. J. Holland began with the evolutionary algorithms first used by Nils Nils Dallman applied to evolutionary simulations. In the 1960s and early 1970s, evolutionary algorithms became widely recognized as optimization methods. Ingo Rechenberg and his group were able to solve complex engineering problems through evolution strategies as documented in his 1971 PhD thesis and the resulting 1973 book. John Holland was highly influential during the 1970s.

In 1964, Lawrence J. Fogel, one of the earliest practitioners of the GP methodology, applied evolutionary algorithms to the problem of discovering finite-state automata. Later GP-related work grew out of the learning classifier system community, which developed sets of sparse rules describing optimal policies for Markov decision processes. The first statement of modern "tree-based" Genetic Programming (that is, procedural languages organized in tree-based structures and operated on by suitably defined GA operators) was given by Michael L. Cramer (1985).[11] This work was later greatly expanded by John R. Koza, a main proponent of GP who has pioneered the application of genetic programming in various complex optimization and search problems.[12]

In the 1990s, GP was mainly used to solve relatively simple problems because it is very computationally intensive. Recently GP has produced many novel and outstanding results in areas such as quantum computing, electronic design, game playing, sorting, and searching, due to improvements in GP technology and the exponential growth in CPU power.[13] These results include the replication or development of several post-year-2000 inventions. GP has also been applied to evolvable hardware as well as computer programs.

Developing a theory for GP has been very difficult and so in the 1990s GP was considered a sort of outcast among search techniques. But after a series of breakthroughs in the early 2000s, the theory of GP has had a formidable and rapid development. So much so that it has been possible to build exact probabilistic models of GP (schema theories and Markov chain models).[citation needed]

**Chromosome representation**

GP evolves computer programs, traditionally represented in memory as tree structures.[14] Trees can be easily evaluated in a recursive manner. Every tree node has an operator function and every terminal node has an operand, making mathematical expressions easy to evolve and evaluate. Thus traditionally GP favors the use of programming languages that naturally embody tree structures (for example, Lisp; other functional programming languages are also suitable).

Non-tree representations have been suggested and successfully implemented, such as linear genetic programming which suits the more traditional imperative languages [see, for example, Banzhaf et al. (1998)]. The commercial GP software Discipulus, uses AIM, automatic induction of binary machine code[15] to achieve better performance. μGP[16] uses a representation similar to linear GP to generate programs that fully exploit the syntax of a given assembly language.

**Genetic operators**

The main operators used in evolutionary algorithms such as GP are crossover and mutation.

**Crossover**

Crossover is applied on an individual by simply switching one of its nodes with another node from another
Crossover is applied to an individual by simply switching one of its nodes with another node from another individual in the population. With a tree-based representation, replacing a node means replacing the whole branch. This adds greater effectiveness to the crossover operator. The expressions resulting from crossover are very much different from their initial parents.

The following code suggests a simple implementation of individual deformation using crossover:

```java
individual.Children[randomChildIndex] = otherIndividual.Children[randomChildIndex];
```

**Mutation**

Mutation affects an individual in the population. It can replace a whole node in the selected individual, or it can replace just the node's information. To maintain integrity, operations must be fail-safe or the type of information the node holds must be taken into account. For example, mutation must be aware of binary operation nodes, or the operator must be able to handle missing values.

A simple piece of code:

```java
individual.Information = randomInformation();
```

or

```java
individual = generateNewIndividual();
```

**Meta-Genetic Programming**

Meta-Genetic Programming is the proposed meta learning technique of evolving a genetic programming system using genetic programming itself. It suggests that chromosomes, crossover, and mutation were themselves evolved, therefore like their real life counterparts should be allowed to change on their own rather than being determined by a human programmer. Meta-GP was formally proposed by Jürgen Schmidhuber in 1987,[17] but some earlier efforts may be considered instances of the same technique, including Doug Lenat's Eurisko. It is a recursive but terminating algorithm, allowing it to avoid infinite recursion.

Critics of this idea often say this approach is overly broad in scope. However, it might be possible to constrain the fitness criterion onto a general class of results, and so obtain an evolved GP that would more efficiently produce results for sub-classes. This might take the form of a Meta evolved GP for producing human walking algorithms which is then used to evolve human running, jumping, etc. The fitness criterion applied to the Meta GP would simply be one of efficiency.

For general problem classes there may be no way to show that Meta GP will reliably produce results more efficiently than a created algorithm other than exhaustion. The same holds for standard GP and other search algorithms.

**Implementations**

Possibly most used:

- ECI - Evolutionary Computation/Genetic Programming research system
- ECO - Evolutionary Computation Genetic Programming Research System (http://cs.gmu.edu/~eclab/projects/ecj/) (Java)
- Lil-Gp (http://garage.cse.msu.edu/software/lil-gp/) Genetic Programming System (C).
- Beagle - Open BEAGLE, a versatile EC framework (http://beagle.sf.net/) (C++ with STL)
- EO Evolutionary Computation Framework (http://eodev.sourceforge.net/) (C++ with static polymorphism)

Others:
- BraneCloud Evolution - Fork of ECJ for .NET 4.0 (http://branecloud.codeplex.com/) (C#)
- RobGP - Robust Genetic Programming System (http://robgp.sourceforge.net/about.php) (C++)
- GPC++ - Genetic Programming C++ Class Library (http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/weinbenner/gp.html) (C++)
- TinyGP (http://cswww.essex.ac.uk/staff/sml/gecco/TinyGP.html) (C and Java)
- GenPro - Reflective Object Oriented Genetic Programming (http://code.google.com/p/genpro/) Open Source Framework. Extend with POJO's, generates plain Java code. (Java)
- deap (http://code.google.com/p/deap/) Distributed Evolutionary Algorithms in Python (Python)
- pySTEP (http://pystep.sourceforge.net/) pySTEP or Python Strongly Typed Genetic Programming
- Pyevolve (http://pyevolve.sourceforge.net/) (Python)
- JAGA - Extensible and pluggable open source API for implementing genetic algorithms and genetic programming applications (http://www.jaga.org) (Java)
- RMIT GP - A Genetic Programming Package with support for Automatically Defined Functions (http://goanna.cs.rmit.edu.au/~vc/rmitgp/) (C++)
- GPE - Framework for conducting experiments in Genetic Programming (http://gpe.sourceforge.net/) (.NET)
- DOPF - simple Genetic Programming research system (http://dgpf.sourceforge.net/) (Java)
- n-genes - Java Genetic Algorithms and Genetic Programming (stack oriented) framework (http://cui.unige.ch/spc/tools/n-genes/) (Java)
- PMDGP - object oriented framework for solving genetic programming problems (http://pmdgp.sourceforge.net/) (C++)
- DRP - Directed Ruby Programming, Genetic Programming & Grammatical Evolution Library (http://drp.rubyforge.org) (Ruby)
- GLAB - A Genetic Programming Toolbox for MATLAB (http://glab.sourceforge.net) (MATLAB)
- GPTIPS - Genetic Programming Tool for MATLAB (http://sites.google.com/site/gptips4matlab/) - aimed at performing multigene symbolic regression (MATLAB)
- PerIGP - Grammar-based genetic programming in Perl (http://perlgp.org) (Perl)
- Discipulus - Commercial Genetic Programming Software from RML Technologies, Inc (http://www.rmltech.com) (Generates code in most high level languages)
- GALib - Object oriented framework with 4 different GA implementations and 4 representation types (arbitrary derivations possible) (http://lancet.mit.edu/ga/) (C++)
- Java GALib - Source Forge open source Java genetic algorithm library, complete with Javadocs and examples (see bottom of page) (http://www.softtechdesign.com/GA/EvolvingABetterSolution-GA.html) (Java)
- LAGEP. Supporting single/multiple population genetic programming to generate mathematical functions. Open Source, OpenMP used. (http://www.cis.nctu.edu.tw/~gis91815/lagep/lagep.html) (C/C++)
- PushGP, a strongly typed, stack-based genetic programming system that allows GP to manipulate its own code (auto-constructive evolution) (http://hampshire.edu/lspector/push.html) (Java/C++/Javascript/Scheme/Clojure/Lisp)
Groovy Java Genetic Programming (http://jgprog.sourceforge.net/) (Java)
GEVA - Grammatical Evolution in Java (http://ncra.ucd.ie/geva/) (Java)
JGE - Java Grammatical Evolution (http://www.bangor.ac.uk/~eep201/jge/) (Java)
ECF - Evolutionary Computation Framework (http://gp.zemris.fer.hr/ecf/) (C++) different genotypes, parallel algorithms, tutorial
JCLEC - Evolutionary Computation Library in Java (http://jclec.sourceforge.net/) (Java) expression tree encoding, syntax tree encoding
HeuristicLab - A Paradigm-Independent and Extensible Environment for Heuristic Optimization (http://dev.heuristiclab.com/) (C#) rich graphical user interface, open source, plugin-based architecture

See also

- Bio-inspired computing
- Gene expression programming
- Genetic representation
- Grammatical evolution
- Fitness approximation
- Linear genetic programming

References and notes

2. AI, where it has been and where it is going. In IJCAI, pages 1152-1157, 1983.
5. Genetic Programming
7. Genetic Programming
8. Angeline, Subtree crossover: Building block engine or macromutation?, GP 1997, p9-17
13. humancompetitive (http://www.genetic-programming.com/humancompetitive.html)
15. (Peter Nordin, 1997, Banzhaf et al., 1998, Section 11.6.2-11.6.3)
17. ^ 1987 THESIS ON LEARNING HOW TO LEARN, METALEARNING, META GENETIC PROGRAMMING, CREDIT-CONSERVING MACHINE LEARNING ECONOMY (http://www.idsi.ch/~juergen/diploma.html)

Bibliography
