What is Genetic Programming?

One of the central challenges of computer science is to get a computer to do what needs to be done, without telling it how to do it. Genetic programming addresses this challenge by providing a method for automatically creating a working computer program from a high-level problem statement of the problem. Genetic programming achieves this goal of *automatic programming* (also sometimes called *program synthesis* or *program induction*) by genetically breeding a population of computer programs using the principles of Darwinian natural selection and biologically inspired operations. The operations include reproduction, crossover (sexual recombination), mutation, and architecture-altering operations patterned after gene duplication and gene deletion in nature.

Genetic programming is a domain-independent method that genetically breeds a population of computer programs to solve a problem. Specifically, genetic programming iteratively transforms a population of computer programs into a new generation of programs by applying analogs of naturally occurring genetic operations. The genetic operations include crossover (sexual recombination), mutation, reproduction, gene duplication, and gene deletion.

Preparatory Steps of Genetic Programming

The human user communicates the high-level statement of the problem to the genetic programming system by performing certain well-defined preparatory steps.

Preparatory Steps of Genetic Programming

Genetic programming starts from a high-level statement of the requirements of a problem and attempts to produce a computer program that solves the problem.

The human user communicates the high-level statement of the problem to the genetic programming system by performing certain well-defined preparatory steps.

The five major preparatory steps for the basic version of genetic programming require the human user to specify

(1) the set of terminals (e.g., the independent variables of the problem, zero-argument functions, and random constants) for each branch of the to-be-evolved program,

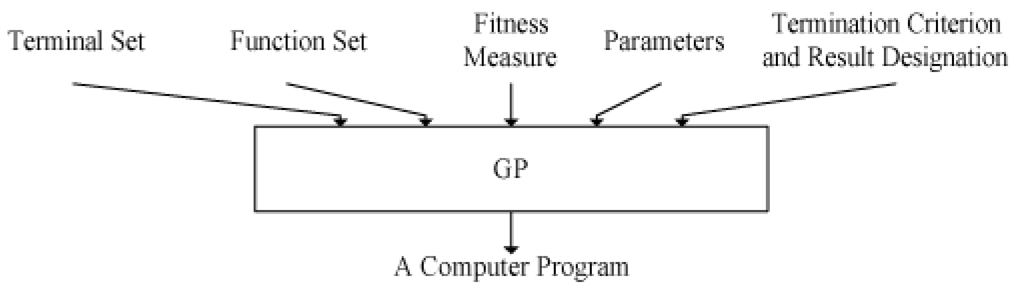
(2) the set of primitive functions for each branch of the to-be-evolved program,

(3) the fitness measure (for explicitly or implicitly measuring the fitness of individuals in the population),

(4) certain parameters for controlling the run, and

(5) the termination criterion and method for designating the result of the run.

The figure below shows the five major preparatory steps for the basic version of genetic programming. The preparatory steps (shown at the top of the figure) are the human-supplied input to the genetic programming system. The computer program (shown at the bottom) is the output of the genetic programming system.



The first two preparatory steps specify the ingredients that are available to create the computer programs. A run of genetic programming is a competitive search among a diverse population of programs composed of the available functions and terminals.

Function Set and Terminal Set

The identification of the function set and terminal set for a particular problem (or category of problems) is usually a straightforward process. For some problems, the function set may consist of merely the arithmetic functions of addition, subtraction, multiplication, and division as well as a conditional branching operator. The terminal set may consist of the program's external inputs (independent variables) and numerical constants. This function set and terminal set is useful for a wide variety of problems (and corresponds to the basic operations found in virtually every general-purpose digital computer).

For many other problems, the ingredients include specialized functions and terminals. For example, if the goal is to get genetic programming to automatically program a robot to mop the entire floor of an obstacle-laden room, the human user must tell genetic programming what the robot is capable of doing. For example, the robot may be capable of executing functions such as moving, turning, and swishing the mop.

If the goal is the automatic creation of a controller, the function set may consist of signal-processing functions that operates on time-domain signals, including integrators, differentiators, leads, lags, gains, adders, subtractors, and the like. The terminal set may consist of signals such as the reference signal and plant output. Once the human user has identified the primitive ingredients for a problem of controller synthesis, the same function set and terminal set can be used to automatically synthesize a wide variety of different controllers.

If a complex structure, such an antenna, is to be designed, it may be desirable to use functions that cause a **turtle** to draw the complex structure.

If the goal is the automatic synthesis of an analog electrical circuit, the function set may enable genetic programming to construct circuits from components such as transistors, capacitors, and resistors. This construction (developmental) process typically starts with a very simple embryonic structure, such as a single modifiable wire. If, additionally, such a function set is geographically aware, a circuit's placement and routing can be synthesized at the same as its topology and sizing. Click here for information about **developmental genetic programming**. Once the human user has identified the primitive ingredients for a problem of circuit synthesis, the same function set and terminal set can be used to automatically synthesize an amplifier, computational circuit, active filter, voltage reference circuit, or any other circuit composed of these ingredients.

Fitness Measure

The third preparatory step concerns the fitness measure for the problem. The fitness measure specifies what needs to be done. The fitness measure is the primary mechanism for communicating the high-level statement of the problem's requirements to the genetic programming system. For example, if the goal is to get genetic programming to automatically synthesize an amplifier, the fitness function is the mechanism for telling genetic programming to synthesize a circuit that amplifies an incoming signal (as opposed to, say, a circuit that suppresses the low frequencies of an incoming signal or a circuit that computes the square root of the incoming signal). The first two preparatory steps define the search space whereas the fitness measure implicitly specifies the search's desired goal.

Control Parameters

The fourth and fifth preparatory steps are administrative. The fourth preparatory step entails specifying the control parameters for the run. The most important control parameter is the population size. In practice, the user may choose a population size that will produce a reasonably large number of generations in the amount of computer time we are willing to devote to a problem (as opposed to, say, analytically choosing the population size by somehow analyzing a problem's fitness landscape). Other control parameters include the probabilities of performing the genetic operations, the maximum size for programs, and other details of the run.

Termination

The fifth preparatory step consists of specifying the termination criterion and the method of 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. In practice, one may manually monitor and manually terminate the run when the values of fitness for numerous successive best-of-generation individuals appear to have reached a plateau. The single best-so-far individual is then harvested and designated as the result of the run.

Running Genetic Programming

After the human user has performed the preparatory steps for a problem, the run of genetic programming can be launched. Once the run is launched, a series of well-defined, problem-independent executional steps (that is, the **flowchart of genetic programming**) is executed.

Click here for an example of an **illustrative run of genetic programming** for a problem of symbolic regression of a quadratic polynomial.

The above five major preparatory steps for the basic version of genetic programming require the human user to specify

(1) the set of terminals (e.g., the independent variables of the problem, zero-argument functions, and random constants) for each branch of the to-be-evolved program,

(2) the set of primitive functions for each branch of the to-be-evolved program,

(3) the fitness measure (for explicitly or implicitly measuring the fitness of individuals in the population),

(4) certain parameters for controlling the run, and

(5) the termination criterion and method for designating the result of the run.

Executional Steps of Genetic Programming

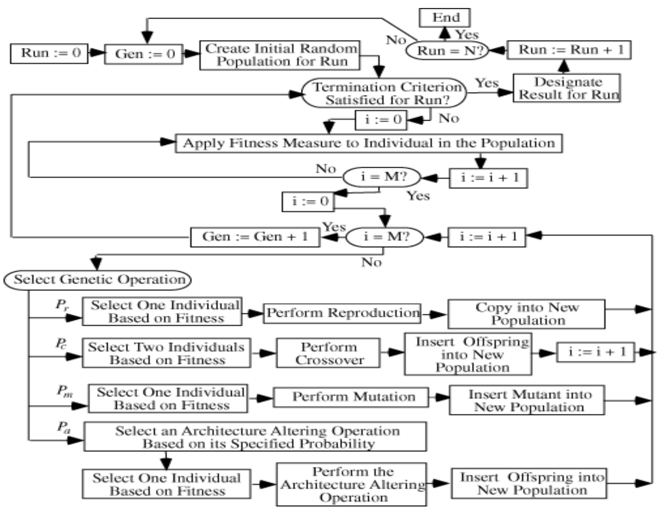
Genetic programming typically starts with a population of randomly generated computer programs composed of the available programmatic ingredients. Genetic programming iteratively transforms a population of computer programs into a new generation of the population by applying analogs of naturally occurring genetic operations. These operations are applied to individual(s) selected from the population. The individuals are probabilistically selected to participate in the genetic operations based on their fitness (as measured by the fitness measure provided by the human user in the third preparatory step). The iterative transformation of the population is executed inside the main generational loop of the run of genetic programming.

Flowchart (Executional Steps) of Genetic Programming

Genetic programming is problem-independent in the sense that the flowchart specifying the basic sequence of executional steps is not modified for each new run or each new problem.

There is usually no discretionary human intervention or interaction during a run of genetic programming (although a human user may exercise judgment as to whether to terminate a run).

The figure below is a flowchart showing the executional steps of a run of genetic programming. The flowchart shows the genetic operations of crossover, reproduction, and mutation as well as the architecture-altering operations. This flowchart shows a two-offspring version of the crossover operation.



End
Yes
No
Run := 0
Gen := 0
Create Initial Random Population for Run
Run = N?
Run := Run + 1
Termination Criterion Satisfied for Run?
Yes
Designate Result for Run
i := 0
No
Apply Fitness Measure to Individual in the Population
No
i = M?
i := i + 1
Yes
i := 0
Gen := Gen + 1
Yes
i = M?
i := i + 1
No
Select Genetic Operation
P_r
Select One Individual Based on Fitness
Perform Reproduction
Copy into New Population
P_c
Select Two Individuals Based on Fitness
Perform Crossover
Insert Offspring into New Population
i := i + 1
P_m
Select One Individual Based on Fitness
Perform Mutation
Insert Mutant into New Population
P_a
Select an Architecture Altering Operation Based on its Specified Probability
Select One Individual Based on Fitness
Perform the Architecture Altering Operation
Insert Offspring into New Population

Overview of Flowchart

Genetic programming starts with an initial population of computer programs composed of functions and terminals appropriate to the problem. The individual programs in the initial population are typically generated by recursively generating a rooted point-labeled program tree composed of random choices of the primitive functions and terminals (provided by the human user as part of the first and second **preparatory steps** of a run of genetic programming). The initial individuals are usually generated subject to a pre-established maximum size (specified by the user as a minor parameter as part of the fourth preparatory step). In general, the programs in the population are of different size (number of functions and terminals) and of different shape (the particular graphical arrangement of functions and terminals in the program tree).

Each individual program in the population is executed. Then, each individual program in the population is either measured or compared in terms of how well it performs the task at hand (using the fitness measure provided in the third preparatory step). For many problems (including all problems in this book), this measurement yields a single explicit numerical value, called *fitness*. The fitness of a program may be measured in many different ways, including, 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 recognizing patterns or classifying objects into classes, the payoff that a game-playing program produces, or the compliance of a complex structure (such as an antenna, circuit, or controller) with user-specified design criteria. The execution of the program sometimes returns one or more explicit values. Alternatively, the execution of a program may consist only of side effects on the state of a world (e.g., a robot’s actions). Alternatively, the execution of a program may produce both return values and side effects.

The fitness measure is, for many practical problems, multiobjective in the sense that it combines two or more different elements. The different elements of the fitness measure are often in competition with one another to some degree.

For many problems, each program in the population is executed over a representative sample of different *fitness cases*. These fitness cases may represent different values of the program’s input(s), different initial conditions of a system, or different environments. Sometimes the fitness cases are constructed probabilistically.

The creation of the initial random population is, in effect, a blind random search of the search space of the problem. It provides a baseline for judging future search efforts. Typically, the individual programs in generation 0 all have exceedingly poor fitness. Nonetheless, some individuals in the population are (usually) more fit than others. The differences in fitness are then exploited by genetic programming. Genetic programming applies Darwinian selection and the genetic operations to create a new population of offspring programs from the current population.

The genetic operations include crossover (sexual recombination), mutation, reproduction, and the architecture-altering operations. These genetic operations are applied to individual(s) that are probabilistically selected from the population based on fitness. In this probabilistic selection process, better individuals are favored over inferior individuals. However, the best individual in the population is not necessarily selected and the worst individual in the population is not necessarily passed over.

After the genetic operations are performed on the current population, the population of offspring (i.e., the new generation) replaces the current population (i.e., the now-old generation). This iterative process of measuring fitness and performing the genetic operations is repeated over many generations.

The run of genetic programming terminates when the termination criterion (as provided by the fifth preparatory step) is satisfied. The outcome of the run is specified by the method of result designation. The best individual ever encountered during the run (i.e., the best-so-far individual) is typically designated as the result of the run.

All programs in the initial random population (generation 0) of a run of genetic programming are syntactically valid, executable programs. The genetic operations that are performed during the run (i.e., crossover, mutation, reproduction, and the architecture-altering operations) are designed to produce offspring that are syntactically valid, executable programs. Thus, every individual created during a run of genetic programming (including, in particular, the best-of-run individual) is a syntactically valid, executable program.

There are numerous alternative implementations of genetic programming that vary from the foregoing brief description.

Creation of Initial Population of Computer Programs

Genetic programming starts with a primordial ooze of thousands of randomly-generated computer programs. The set of functions that may appear at the internal points of a program tree may include ordinary arithmetic functions and conditional operators. The set of terminals appearing at the external points typically include the program's external inputs (such as the independent variables X and Y) and random constants (such as 3.2 and 0.4). The randomly created programs typically have different sizes and shapes. **Click here for animated example of random creation of two illustrative computer programs**.

Main Generational Loop of Genetic Programming

The main generational loop of a run of genetic programming consists of the fitness evaluation, Darwinian selection, and the genetic operations. Each individual program in the population is evaluated to determine how fit it is at solving the problem at hand. Programs are then probabilistically selected from the population based on their fitness to participate in the various genetic operations, with reselection allowed. While a more fit program has a better chance of being selected, even individuals known to be unfit are allocated some trials in a mathematically principled way. That is, genetic programming is not a purely greedy hill-climbing algorithm.

The individuals in the initial random population and the offspring produced by each genetic operation are all syntactically valid executable programs.

After many generations, a program may emerge that solves, or approximately solves, the problem at hand.

Mutation Operation

In the mutation operation, a single parental program is probabilistically selected from the population based on fitness. A mutation point is randomly chosen, the subtree rooted at that point is deleted, and a new subtree is grown there using the same random growth process that was used to generate the initial population. This asexual mutation operation is typically performed sparingly (with a low probability of, say, 1% during each generation of the run). **Click here for animated example of mutation operation**.

Crossover (Sexual Recombination) Operation

In the crossover, or sexual recombination operation, two parental programs are probabilistically selected from the population based on fitness. The two parents participating in crossover are usually of different sizes and shapes. A crossover point is randomly chosen in the first parent and a crossover point is randomly chosen in the second parent. Then the subtree rooted at the crossover point of the first, or receiving, parent is deleted and replaced by the subtree from the second, or contributing, parent. Crossover is the predominant operation in genetic programming (and genetic algorithm) work and is performed with a high probability (say, 85% to 90%). **Click here for animated example of crossover operation**.

Reproduction Operation

The reproduction operation copies a single individual, probabilistically selected based on fitness, into the next generation of the population.

Architecture-Altering Operations

Simple computer programs consist of one main program (called a result-producing branch). However, more complicated programs contain subroutines (also called automatically defined functions, ADFs, or function-defining branches), iterations (automatically defined iterations or ADIs), loops (automatically defined loops or ADLs), recursions (automatically defined recursions or ADRs), and memory of various dimensionality and size (automatically defined stores or ADSs). If a human user is trying to solve an engineering problem, he or she might choose to simply prespecify a reasonable fixed architectural arrangement for all programs in the population (i.e., the number and types of branches and number of arguments that each branch possesses). Genetic programming can then be used to evolve the exact sequence of primitive work-performing steps in each branch.

However, sometimes the size and shape of the solution *is* the problem (or at least a major part of it). Genetic programming is capable of making all architectural decisions dynamically during the run of genetic programming. Genetic programming uses architecture-altering operations to automatically determine program architecture in a manner that parallels gene duplication in nature and the related operation of gene deletion in nature. Architecture-altering operations provide a way, dynamically during the run of genetic programming, to add and delete subroutines and other types of branches to individual programs to add and delete arguments possessed by the subroutines and other types of branches. These architecture-altering operation quickly create an architecturally diverse population containing programs with different numbers of subroutines, arguments, iterations, loops, recursions, and memory and, also, different hierarchical arrangements of these elements. Programs with architectures that are well-suited to the problem at hand will tend to grow and prosper in the competitive evolutionary process, while programs with inadequate architectures will tend to wither away under the relentless selective pressure of the problem's fitness measure. Thus, the architecture-altering operations relieve the human user of the task of prespecifying program architecture.

There are several different architecture-altering operations (described below). They are each applied sparingly during the run (say, with a probability of 1/2% of 1% on each generation).

The subroutine duplication operation duplicates a preexisting subroutine in an individual program gives a new name to the copy and randomly divides the preexisting calls to the old subroutine between the two. This operation changes the program architecture by broadening the hierarchy of subroutines in the overall program. As with gene duplication in nature, this operation preserves semantics when it first occurs. The two subroutines typically diverge later, sometimes yielding specialization. **Click here for animated example of the subroutine duplication operation.**

The argument duplication operation duplicates one argument of a subroutine, randomly divides internal references to it, and preserves overall program semantics by adjusting all calls to the subroutine. This operation enlarges the dimensionality of the subspace on which the subroutine operates. **Click here for animated example of the argument duplication operation**.

The subroutine creation operation can create a new subroutine from part of a main result-producing branch thereby deepening the hierarchy of references in the overall program, by creating a hierarchical reference between the main program and the new subroutine. The subroutine creation operation can also create a new subroutine from part of an existing subroutine further deepening the hierarchy of references, by creating a hierarchical reference between a preexisting subroutine and a new subroutine and a deeper and more complex overall hierarchy. **Click here for animated example of the subroutine creation operation.**

The architecture-altering operation of subroutine deletion deletes a subroutine from a program thereby making the hierarchy of subroutines either narrower or shallower. **Click here for animated example of the subroutine deletion operation.**

The argument deletion operation deletes an argument from a subroutine thereby reducing the amount of information available to the subroutine, a process that can be viewed as generalization. **Click here for animated example of the argument deletion operation**.

Other architecture-altering operations add and delete automatically defined iterations, automatically defined loops, automatically defined recursions, and automatically defined stores (memory).

Click here for an example of an **illustrative run of genetic programming** for a problem of symbolic regression of a quadratic polynomial

The executional steps of genetic programming (that is, the flowchart of genetic programming) are as follows:

(1) Randomly create an initial population (generation 0) of individual computer programs composed of the available functions and terminals.

(2) Iteratively perform the following sub-steps (called a *generation*) on the population until the termination criterion is satisfied:

(a) Execute each program in the population and ascertain its fitness (explicitly or implicitly) using the problem's fitness measure.

(b) Select one or two individual program(s) from the population with a probability based on fitness (with reselection allowed) to participate in the genetic operations in (c).

(c) Create new individual program(s) for the population by applying the following genetic operations with specified probabilities:

(i) *Reproduction:* Copy the selected individual program to the new population.

(ii) *Crossover:* Create new offspring program(s) for the new population by recombining randomly chosen parts from two selected programs.

(iii) *Mutation:* Create one new offspring program for the new population by randomly mutating a randomly chosen part of one selected program.

(iv) *Architecture-altering operations:* Choose an architecture-altering operation from the available repertoire of such operations and create one new offspring program for the new population by applying the chosen architecture-altering operation to one selected program.

(3) After the termination criterion is satisfied, the single best program in the population produced during the run (the best-so-far individual) is harvested and designated as the result of the run. If the run is successful, the result may be a solution (or approximate solution) to the problem.

• For information about the field of genetic programming in general, visit **www.genetic-programming.org**

• The home page of **John R. Koza at Genetic Programming Inc.** (including online versions of most papers) and the home page of **John R. Koza at Stanford University**

• Information about the 1992 book ***Genetic Programming: On the Programming of Computers by Means of Natural Selection***, the 1994 book ***Genetic Programming II: Automatic Discovery of Reusable Programs***, the 1999 book ***Genetic Programming III: Darwinian Invention and Problem Solving***, and the 2003 book ***Genetic Programming IV: Routine Human-Competitive Machine Intelligence***. Click here to read **chapter 1 of *Genetic Programming IV* book in PDF format.**

• For information on 3,198 papers (many on-line) on genetic programming (as of June 27, 2003) by over 900 authors, see **William Langdon's bibliography on genetic programming**.

• For information on the **Genetic Programming and Evolvable Machines journal** published by Kluwer Academic Publishers

- For information on the Genetic Programming book series from Kluwer Academic Publishers, see the **Call For Book Proposals**

- For information about the annual **Genetic and Evolutionary Computation (GECCO) conference** (which includes the annual GP conference) to be held on June 26–30, 2004 (Saturday – Wednesday) in Seattle and its sponsoring organization, the International Society for Genetic and Evolutionary Computation (**ISGEC**). For information about the annual **NASA/DoD Conference on Evolvable Hardware Conference (EH)** to be held on June 24-26 (Thursday-Saturday), 2004 in Seattle. For information about the annual **Euro-Genetic-Programming Conference** to be held on April 5-7, 2004 (Monday – Wednesday) at the University of Coimbra in Coimbra Portugal.