Genetic Programming

For Dummies

Features, merits, weaknesses, pseudo-code and a simple worked example

A Report

Namir Shammas · namirshammas.com · September 2026

Contents

Contents 2
Introduction 4
Chapter 1: What Genetic Programming Is 5
Evolution as a way of searching 5
Genetic algorithms versus genetic programming 5
A little history 5
Chapter 2: How Genetic Programming Works 6
Programs as trees 6
The building blocks 6
Scoring: fitness 6
Picking parents: selection 7
Making children: crossover 7
Making children: mutation 7
The loop 7
Chapter 3: Features of a Genetic Programming System 8
Ramped half-and-half initialization 8
Size and depth limits 8
Parsimony pressure 8
Constants 8
Automatically defined functions 8
Multiple runs and seeds 8
Output you can use 9
Chapter 4: The Merits 10
Chapter 5: The Weaknesses 11
Chapter 6: The Algorithm in Pseudo-code 12
High-level pseudo-code 12
Mid-level pseudo-code 12
Chapter 7: A Simple Step-by-Step Example 15
Step 1: The data 15
Step 2: The building blocks 15
Step 3: A random crowd (generation 0) 15

Step 4: Pick parents 16
Step 5: Make children (generation 1) 16
Step 6: Generation 2 17
Step 7: Stop and report 17
Chapter 8: The Part of Tens — Ten Things to Remember 19
Where to Go from Here 20

Introduction

Welcome to Genetic Programming For Dummies. Genetic programming is a way of getting a computer to write its own small programs, or formulas, by imitating evolution: make a crowd of random programs, keep the ones that work best, breed them, and repeat. It sounds too simple to work, and for thirty years it has worked anyway, on everything from curve fitting to circuit design to controlling robots.

You don't need a computer science degree to follow this report. If you can read a formula like (x + 1) × 3 and you understand what "closer to the right answer" means, you have what you need. The technical details that can't be avoided are fenced off in boxes you can skip the first time through.

How this report is organized

Chapter 1 says what genetic programming is and where it came from. Chapter 2 opens the hood: programs as trees, scoring, selection, crossover and mutation. Chapter 3 lists the features you will meet in a real system. Chapters 4 and 5 give the merits and the weaknesses plainly. Chapter 6 shows the algorithm as pseudo-code, first in seven lines and then in enough detail to program from. Chapter 7 works a small example by hand, tree by tree, from random guesses to the right answer in two generations. Chapter 8 is the Part of Tens.

Icons used in this report

> **TIP** A shortcut, a good habit, or a way to save yourself some grief.

> **REMEMBER** An idea the rest of the report builds on.

> **WARNING** A place where people commonly go wrong, or where the method can bite you.

> **TECHNICAL STUFF** Details for the curious. Skip these on the first pass; nothing later depends on them.

Chapter 1: What Genetic Programming Is

In this chapter: evolution as a search method; what makes genetic programming different from a genetic algorithm; and a little history.

Evolution as a way of searching

Suppose you want a formula that fits some data, but you have no idea what shape it should be. You could try formulas one at a time, forever. Or you could do what nature does: make lots of candidates, let the bad ones die, let the good ones have children that mix their features, and wait. Over many rounds, the crowd of candidates gets better without anyone designing anything. That, in one paragraph, is evolutionary computation.

Genetic algorithms versus genetic programming

A genetic algorithm (GA) evolves a fixed list of numbers or bits: the settings of a machine whose design is already fixed. It can tune the knobs, but it cannot add a knob. Genetic programming (GP) evolves the machine itself. Its individuals are programs, usually written as trees of operations, and they can be any shape and any size. That freedom is what makes GP powerful, and it is also the source of most of its headaches, as Chapter 5 explains.

> **REMEMBER** A genetic algorithm evolves the numbers inside a fixed design. Genetic programming evolves the design. If you remember one sentence from this report, make it that one.

A little history

The idea of evolving programs was floated in the 1980s, but genetic programming as a field dates from John Koza's 1992 book Genetic Programming, which showed that tree-shaped programs, bred by swapping branches, could solve a startling range of problems. Koza's follow-up books added automatically defined functions and, later, showed GP reinventing patented electronic circuits. Today GP underlies most symbolic regression tools, and its ideas live on inside newer methods that use neural networks or language models to guide the breeding.

Chapter 2: How Genetic Programming Works

In this chapter: programs as trees; the building blocks; scoring; picking parents; and the two ways of making children.

Programs as trees

GP stores a program as a tree. The top node is an operation; its children are the things the operation acts on; and so on down to the leaves, which are variables or numbers. The formula (x + 1) × 3 is a tree with × at the top, a + branch on the left holding x and 1, and a 3 on the right.

```
        *
       / \
      +   3        reads as:  (x + 1) * 3
     / \
    x   1
```

Why trees? Because any branch of a tree is itself a complete little program, so you can cut a branch off one tree and graft it onto another and the result is always a legal program. No other representation makes breeding this easy.

The building blocks

Before a run you choose two lists. The function set holds the operations: +, −, ×, ÷ and perhaps sqrt, log, sin. The terminal set holds the leaves: your input variables and some constants. GP will combine these in every way it can think of, so keep the lists small and relevant.

> **TECHNICAL STUFF** Two properties matter. Closure: every function must be able to accept whatever any other function or terminal produces, so any tree is a valid program; division is usually "protected" to return something sensible when the divisor is zero. Sufficiency: the building blocks must be capable of expressing a solution at all; if the answer needs a logarithm and you didn't include one, GP can only approximate.

Scoring: fitness

Each program is run on the data and scored. For curve fitting the score is the error, usually the mean of the squared differences between what the program predicts and what the data say. Lower is better. Other problems use other scores: for a robot controller, distance traveled; for a classifier, how many cases it gets right. GP does not care what the score is, as long as it can compare two programs.

Picking parents: selection

Parents are chosen with a bias toward the better programs, but not only the best. The usual method is a tournament: pick a few programs at random and let the best of them be the parent. A small tournament is gentle and keeps the crowd diverse; a big one is harsh and converges fast. The very best program is usually copied straight into the next generation unchanged, so that it can never be lost. That is called elitism.

Making children: crossover

Crossover takes two parents, picks a random branch in each, and swaps them. The child has most of one parent and a piece of the other. This is GP's main engine, applied to most of the children in every generation.

```
Parent 1:  (x + 1) * 3        Parent 2:  x + x

Pick the branch "1" in parent 1 and the branch "x + x" in parent 2.
Graft parent 2's branch where parent 1's was:

Child:     (x + (x + x)) * 3
```

Making children: mutation

Mutation takes one parent, picks a random branch, throws it away and grows a small random branch in its place. Mutation is the way brand-new material enters the crowd; crossover can only shuffle what is already there.

The loop

Put it together: make a random crowd, score everyone, keep the champion, choose parents by tournament, make a new crowd by crossover and mutation, and repeat until the champion is good enough or you run out of patience. Chapter 6 writes this out properly, and Chapter 7 runs it by hand.

Chapter 3: Features of a Genetic Programming System

In this chapter: the features and settings you will find on any real GP system, and what they are for.

Ramped half-and-half initialization

The first crowd matters, because crossover can only recombine what is there. Koza's recipe makes half the trees "full" (every branch grows to the same depth) and half "grown" (branches stop at random), with depths ramped from small to large. The result is a crowd with trees of every shape and size.

Size and depth limits

A GP system always has a maximum depth or size, because without one the trees keep growing (see "bloat" in Chapter 5). Koza used depth 17. A child that breaks the limit is thrown away or replaced by a copy of a parent.

Parsimony pressure

Many systems add a small penalty for size to the score, so that between two equally accurate programs the shorter wins. It is the simplest cure for bloat and it makes the results more readable.

Constants

Classical GP handles numbers clumsily: a random constant is dropped into a leaf and never changes unless mutation replaces it. Better systems tune the constants of each program with a numerical optimizer, or fit the outermost scale and offset by least squares ("linear scaling"). If your problem has important constants, this feature matters more than any other.

Automatically defined functions

A program can be given branches that act as reusable subroutines, which the main branch may call several times. This lets GP discover a useful piece once and reuse it, rather than rediscovering it in every place it is needed.

Multiple runs and seeds

GP is random. A serious system lets you fix the random seed for reproducibility and makes it easy to run many times and compare. Success is usually reported as "solved in 8 of 10 runs," not as a single result.

Output you can use

The winning tree is simplified (x × 1 + 0 becomes x) and printed as a readable formula, a line of code, or both.

Chapter 4: The Merits

In this chapter: why anyone bothers with a method this expensive.

It needs no assumptions about shape. Where every other fitting method starts from a formula you chose, GP starts from building blocks and lets the data decide. When the true relationship is something you would never have guessed, this is decisive.

Its answers are readable. A GP result is a formula or a small program. You can read it, plot it, simplify it, check it against the textbook, and put it in a spreadsheet. A neural network gives you a million numbers; GP gives you a line.

Any score will do. The fitness function can be anything you can compute, including things with no derivative and no formula, like "how far did the robot get?" Gradient-based methods can't do that.

It is robust. Crossover always makes a legal program, the loop tolerates noise and irrelevant variables, and a working GP fits in a few hundred lines of code. Every program is scored independently, so it parallelizes for free.

You get a family of answers, not one. Every GP run leaves behind a ladder of programs trading accuracy for size, and the whole ladder is often more informative than the single best rung.

> **TIP** Look at the best program of every size, not just the most accurate one. The place where adding size stops buying accuracy is usually where the true answer lives.

Chapter 5: The Weaknesses

In this chapter: the ways GP will frustrate you, and what to do about each.

It is slow. A run scores thousands of programs on every data row, generation after generation. Minutes are normal; hours are common for hard problems.

It bloats. Left alone, trees grow bigger every generation whether or not the extra size buys accuracy. This is not a bug; it is a consequence of how crossover works, because big trees have more places where a cut does no harm, so they survive breeding better. Without size limits or parsimony pressure the winning program becomes unreadable.

> **WARNING** If the answer GP gives you is twenty times longer than you expected, it is bloat, not insight. Simplify it, look at smaller programs on the accuracy ladder, and turn on parsimony pressure next time.

It is random. Two runs give two answers. One run proves nothing; report what many runs agree on.

It is clumsy with numbers. Classical GP can find the shape $x \times \log(y)$ and still get the coefficient wrong, because nothing tunes it. Use a system that optimizes constants, or refit the discovered structure with ordinary regression afterwards.

It overfits small data with enthusiasm. Give GP ten points and a rich function set and it will find a program that passes through all ten, whether or not that program means anything. Hold some data back and test on it.

Most children are worse than their parents. Crossover swaps branches blindly, without regard to what they compute, so a good branch is often dropped into a place where it makes no sense. GP makes progress through the few children that happen to be better, not through any cleverness in the operator.

It has many knobs. Population size, generations, function set, depth limit, tournament size, operator rates: they interact, and there is little theory to say how to set them. Start from defaults and change one thing at a time.

It gives you no statistics. A formula found by search comes with no confidence intervals or p-values. If you need them, refit the structure with a conventional method.

Chapter 6: The Algorithm in Pseudo-code

In this chapter: the loop in seven lines, and then in enough detail to program from.

High-level pseudo-code

```
GENETIC PROGRAMMING, IN SEVEN LINES


1. Choose the building blocks: the operations and the leaves.
2. Make a crowd of random program trees.
3. Run every program on the data and score it.
4. Pick parents by tournament, favoring the better scores.
5. Make children by swapping branches between parents (crossover)
   and by replacing random branches with new random ones (mutation).
6. Keep the champion; replace the crowd with the children.
7. Go back to step 3 until the champion is good enough.
```

Mid-level pseudo-code

```
INPUT:   the data; Functions with their number of arguments; Terminals;
         PopSize, MaxGenerations, MaxDepth, pCrossover, pMutation, TournamentSize
OUTPUT:  the best program found and its score

RandomTree(depth, full):
    if depth = 0 or (not full and a coin flip says "leaf"):
        return a random terminal
    pick a random function f
    return a node f whose children are RandomTree(depth - 1, full), one per argument

InitialPopulation:
    for i = 1 to PopSize:
        depth <- a value ramped between 2 and MaxInitialDepth
        full  <- true for even i, false for odd i          (ramped half-and-half)
        Pop[i] <- RandomTree(depth, full)

Score(tree):
    for each data row: prediction <- Evaluate(tree, row)
    if any prediction is NaN or infinite: return a huge number
    return mean of (target - prediction)^2   (+ optional penalty * size of tree)

Tournament(Pop, Scores):
    pick TournamentSize trees at random; return the one with the best score

Crossover(p1, p2):
    b1 <- a random node in p1 ; b2 <- a random node in p2
    child <- copy of p1 with the branch at b1 replaced by the branch at b2
    if depth(child) > MaxDepth: child <- copy of p1
    return child

Mutation(p):
    b <- a random node in p
    child <- copy of p with the branch at b replaced by RandomTree(2, false)
    if depth(child) > MaxDepth: child <- copy of p
    return child

Main:
    Pop <- InitialPopulation
    for generation = 0 to MaxGenerations:
        Scores <- Score of every tree in Pop
        best <- the tree with the smallest score ; remember it if it is the best ever
        if its score is small enough: stop
        New <- [best]                                      (elitism)
        while New has fewer than PopSize trees:
            r <- random number in [0, 1)
            if r < pCrossover:                add Crossover(Tournament, Tournament)
            else if r < pCrossover + pMutation: add Mutation(Tournament)
            else:                             add a copy of Tournament
        Pop <- New
    return Simplify(best ever), its score
```

> **TECHNICAL STUFF** Typical settings: population 100 to 1000, 50 to 500 generations, crossover 0.8 to 0.9, mutation 0.05 to 0.2, tournament size 3 to 7, maximum depth 8 to 17. Koza chose crossover points 90 percent of the time at interior nodes so that whole sub-formulas get swapped rather than single leaves.

Chapter 7: A Simple Step-by-Step Example

In this chapter: five data points, six random trees, and two generations of breeding, done by hand with every number shown.

Step 1: The data

You have measured y at x = 0, 1, 2, 3, 4 and got the values below. (They come from y = 2x + 1, but pretend you don't know that.)

x	0	1	2	3	4
y	1	3	5	7	9

Step 2: The building blocks

Functions: +, −, ×. Terminals: x and the constants 1, 2, 3. Score: mean squared error (MSE) over the five points, so a perfect program scores 0. Crowd size: six. Tournaments of two. The champion is always kept.

Step 3: A random crowd (generation 0)

#	Tree	Predictions at x = 0…4	MSE
A	x + 3	3, 4, 5, 6, 7	2.0
B	x × x	0, 1, 4, 9, 16	11.8
C	2	2, 2, 2, 2, 2	17.0
D	x + 1	1, 2, 3, 4, 5	6.0
E	x + x	0, 2, 4, 6, 8	1.0
F	x × 3	0, 3, 6, 9, 12	3.0

Where does an MSE come from? Take E, x + x. Its predictions 0, 2, 4, 6, 8 miss the targets 1, 3, 5, 7, 9 by exactly 1 each time. Squared, that is 1, 1, 1, 1, 1; the mean is 1.0. E has the right slope and is missing the "+ 1". A, x + 3, has the wrong slope but is close over this short range.

Drawn as trees, the two that matter most look like this:

```
E:   +          A:   +
    / \             / \
   x   x           x   3
```

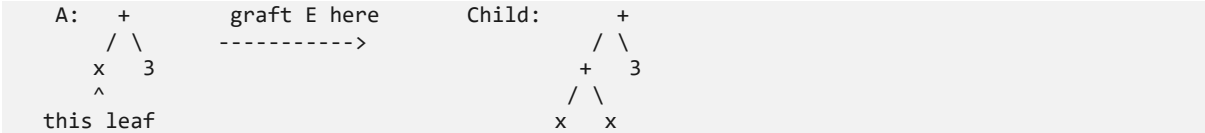
Step 4: Pick parents

Tournaments of two. Suppose the random draws are E vs C (E wins), A vs B (A wins), B vs C (B wins), F vs C (F wins), D vs C (D wins), and E vs D (E wins). The parents are E, A, B, F, D, E. E was picked twice and C, the worst program, never, which is what selection is supposed to do.

Step 5: Make children (generation 1)

The champion E is copied unchanged. Then:

Crossover of E and A. The random branch chosen in E is the whole tree, x + x. The random branch chosen in A is the leaf x. Replace A's x by E's tree: the child is (x + x) + 3, predictions 3, 5, 7, 9, 11, MSE 4.0. Right slope, wrong intercept.



Crossover of B and D. Branch in B: the whole tree x × x. Branch in D: the leaf x. Child: (x × x) + 1, predictions 1, 2, 5, 10, 17, MSE 14.8. Worse than both parents.

Crossover of F and D. Branch in F: the leaf x. Branch in D: the whole tree x + 1. Child: (x + 1) × 3, predictions 3, 6, 9, 12, 15, MSE 18.0. Also worse.

Mutation of E. Branch chosen: the whole tree. New random branch grown: (x + x) − x. Child: (x + x) − x, which is just x, predictions 0, 1, 2, 3, 4, MSE 11.0. A step backward.

Crossover of B and D again, with different random branches. Branch in B: the leaf x on the right. Branch in D: the whole tree x + 1. Child: x × (x + 1), predictions 0, 2, 6, 12, 20, MSE 29.8.

#	Tree	MSE	Where it came from
E	x + x	1.0	champion, copied unchanged
G	(x + x) + 3	4.0	crossover: E grafted into A
H	(x × x) + 1	14.8	crossover: B grafted into D
I	(x + 1) × 3	18.0	crossover: D grafted into F
J	(x + x) − x	11.0	mutation of E
K	x × (x + 1)	29.8	crossover: D grafted into B

REMEMBER Four of the five children are worse than their parents, and the champion did not improve. This is a completely normal generation. The one useful thing that happened is that G, (x + x) + 3, now carries the right slope together with a spare leaf where the intercept should be.

Step 6: Generation 2

Tournaments now favor E (1.0) and G (4.0). Suppose the parents drawn are G, D, E, G.

Crossover of G and D. The random branch in G is the leaf 3. The random branch in D is the leaf 1. Replace the 3 by the 1:

```
G:      +          swap leaf        Child:      +
       / \        ---------->                  / \
      +   3                                   +   1
     / \                                     / \
    x   x                                   x   x

Child:  (x + x) + 1   ->  predictions 1, 3, 5, 7, 9   ->  MSE 0.0
```

That is the answer. The other children of this generation, for the record: crossover of E and G with the whole of D's tree, giving (x + x) + (x + 1), MSE 6.0; mutation of G replacing the 3 by a 2, giving (x + x) + 2, MSE 1.0; and a copy of E.

#	Tree	MSE
E	x + x	1.0 (champion carried over)
L	(x + x) + 1	0.0 ← the answer
M	(x + x) + (x + 1)	6.0
N	(x + x) + 2	1.0

Step 7: Stop and report

The champion's error is zero, so the run stops. The simplifier turns (x + x) + 1 into 2x + 1, and that is the rule the data came from. Two generations were enough because the crowd happened to contain both pieces the answer needed, x + x for the slope and a constant 1 as a leaf in another tree, and one crossover put them together.

WARNING Real runs are not this tidy. With several variables, a function set that includes sqrt and log, and noisy data, expect hundreds of generations, a champion much bigger than the true formula, and a final answer that fits the training points beautifully and the next batch of data less well. The mechanism is exactly what you just saw; there is simply far more haystack.

TIP When you run GP on your own data, set the random seed, run it five times, keep back a fifth of the data for testing, and turn on parsimony pressure. Those four habits prevent most of the disappointments this method can deliver.

Chapter 8: The Part of Tens — Ten Things to Remember

- Genetic programming evolves the design, not just the numbers inside a fixed design.
- Programs are trees, and any branch is a complete little program. That is why cutting and grafting always gives a legal result.
- The loop is: random crowd, score, tournament, crossover and mutation, keep the champion, repeat.
- Crossover swaps branches between parents; mutation grows a new random branch. Crossover shuffles, mutation invents.
- Most children are worse than their parents. Progress comes from selection keeping the rare improvement.
- Keep the function set small. Every extra operation makes the haystack bigger.
- Trees bloat. Use a depth limit and parsimony pressure, and always simplify the answer.
- Classical GP is clumsy with constants. Use a system that tunes them, or refit the structure afterwards.
- One run is an anecdote. Run several, fix the seed, and report what they agree on.
- A program that fits the data is not automatically the right program. Test on data it has never seen and check it against what you know.

Where to Go from Here

John Koza's Genetic Programming (MIT Press, 1992) is the founding text and still very readable in its early chapters. A Field Guide to Genetic Programming by Poli, Langdon and McPhee (2008) is free online and is the best modern introduction. For hands-on experiments, gplearn (Python) is small and clear, PySR (Python/Julia) is fast and modern, and HeuristicLab (Windows) has a graphical interface. Gene expression programming, a close cousin that stores trees as fixed-length strings and never bloats, is described in Cândida Ferreira's 2001 paper and her 2006 book, and is a natural next topic once GP makes sense.