Measuring Failed Disruption Propagation in Genetic Programming

William B. Langdon and Afnan Al-Subaihin and David Clark
W.Langdon@cs.ucl.ac.uk,a.alsubaihin@ucl.ac.uk,david.clark@ucl.ac.uk
Department of Computer Science, University College London, London, UK

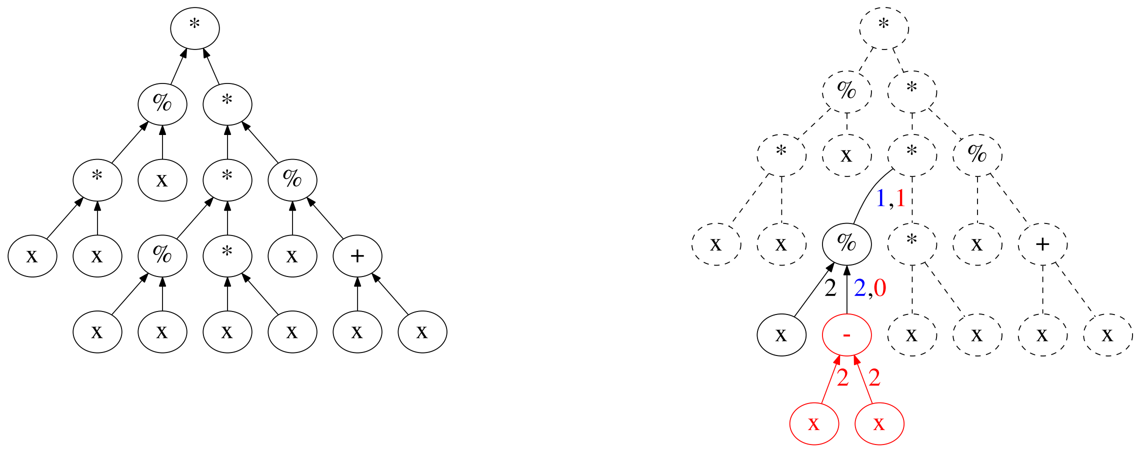

Figure 1: *Left:* example tree from Koza GP video [16, 11 mins 10 secs]. Arrows show information flow from leafs, x, to root node (top oval). Root gives output of program. *Right:* example of failed disruption propagation on test case x=2 with code change (x⇒(- x x) red). Disruption only reaches 4th level so evaluation at root is identical. Old eval blue(left), new red(right) on arrows, unchanged dashed. Incremental evaluation stops at 2nd protected division % as it gives same value (1) in parent and child.

ABSTRACT

Information theory explains the robustness of deep GP trees, with on average up to 83.3% of crossover run time disruptions failing to propagate to the root node, and so having no impact on fitness, leading to phenotypic convergence. Monte Carlo simulations of perturbations covering the whole tree demonstrate a model based on random synchronisation of the evaluation of the parent and child which cause parent and offspring evaluations to be identical. This predicts the effectiveness of fitness measurement grows slowly as O(log(n)) with number n of test cases. This geometric distribution model is tested on genetic programming symbolic regression.

CCS CONCEPTS

• **Computing methodologies** → **Genetic programming**.

KEYWORDS

mutational robustness, antifragile correctness attraction, SBSE, software resilience, information theory, entropy

ACM Reference Format:
William B. Langdon and Afnan Al-Subaihin and David Clark. 2022. Measuring Failed Disruption Propagation in Genetic Programming. In *Genetic and Evolutionary Computation Conference (GECCO '22), July 9–13, 2022, Boston, MA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3512290.3528738

1 INTRODUCTION

1.1 Failed Disruption Propagation

Petke, Clark & Langdon [32] define Failed Disruption Propagation (FDP) as when software execution is disrupted but later we do not see evidence of the disruption. Petke et al. [32] showed that FDP underlies the various types of robustness in software engineering. By **disruption** we mean when bugs, run time errors, mutations, radiation, electromagnetic interference, malicious external noise, etc., cause an internal difference to the smooth running of the program. For example, in genetic programming [14], [2], [27], [33] crossover (e.g. red subtree in Figure 1) potentially changes the evaluation on every fitness test case. Notice however that in Figure 1 the disruption on test case x=2 caused by replacing x=2 by (-x x) = 0 **fails** to **propagate** to the root node.

In the example shown in Figure 1 disruption almost always occurs, however a change at run time need not always happen. For example, a bug may not be executed, or even if the buggy code is used, it may be on a particular test it just happens to generate the same answer as the working code (e.g. x=0 in Figure 1). Even if a change is made, In [32] we argue that in many cases this disruption fails to propagate to the program's outputs. In deterministic programs information theory gives sound reasons why as the disruption spreads through the executing program intermediate computations not only progressively dilute the disruption but may prevent it from having a measurable external impact (see Section 2.3 and Figure 4).

In [32] we do not claim to make programs bug free [28], but instead that in many cases the impact of the bug cannot be seen outside the program. Petke, Clark & Langdon [32] unifies several ideas in software engineering: e.g. correctness attraction [7], antifragile software [31], not fragile [23], equivalent mutants [10], mutational robustness [35], neutrality [11], coincidental correctness [1], chaos engineering [3] and software robustness [37], but our focus is on evolution [8], particularly genetic programming.

Often in the software engineering literature the various causes and outcomes are confused: Is there a bug? Was it triggered? Is the system now under attack? Did the error cause a faulty result? Will it give an error the next time the program is used? Therefore we deliberately separate the cause of the disruption from its propagation by ensuring we know where the cause is, guaranteeing it is executed and then **measuring** how it propagates. (In Sections 3 and 6 we test every point in the GP trees.) We study failed disruption in typical symbolic regression GP trees and in high order polynomials. As is common in genetic programming, they do not have side effects, allowing us to measure FDP in highly nested pure structures.

In Sections 1.2 and 1.3 we complete our introduction. Then Section 2 discusses information theory in relation to failed disruption propagation and GP and the background to our Monte Carlo calculations (Section 3). In Section 3 we generate a wide range of sizes (1 to 20 000 001) for GP like trees and show the effect of assuming a geometric distribution on the distance disruption travels before FDP occurs. This allows us to model the whole GP search space rather than just the fit trees that GP evolves. To test both rapidly quenching (1.0) and very long acting disruptions (130), we use geometric distributions with a wide range of plausible means. To keep the calculations manageable we selected five means in this range. Indeed Section 3 calculates more than six billion probabilities for these Monte Carlo simulations. From these simulations, Section 4 shows crossover or mutation deeper than two or three times the geometric distribution's mean have little chance of changing the tree's evaluation. Section 5 shows, without additional knowledge, adding extra tests to the test set only increases its ability to measure the fitness impact of deep genetic changes slowly $\leq\propto \log(n)$. Section 6 shows in most of the trees failed disruption propagation does indeed resemble the geometric distribution, with means (1..120) depending on test value. However close to the root node, the geometric model does not hold, but in large evolved trees this is only as small fraction of the tree. Of course not all systems depend upon a single root node.

1.2 Side Effects and Other Types of GP

The vast majority of GP systems evolve programs which return their answer as the program terminates. Although the model presented below deals with trees, this input-execute-result-stop mode of working is found in both tree and linear GP [2]. Some GP systems co-evolve multiple trees, e.g. ADFs [15] and multiple classifier systems [22]. However, although they have multiple evolved root nodes, typically the different trees' answers are combined into a single answer, either by an externally imposed framework (e.g. voting or weights) or, in the case of ADFs, by evolved code. Common alternatives include agent control strategies, in which, as the evolved program runs, it reads information from the environment (e.g. no food ahead) and sends instructions (e.g. turn left). For example, the artificial ant benchmark [14],[26],[27] and robot soccer [29],[6]. Teller's "any time" algorithm requires the GP program to give its result at any point and so disconnects result giving from program termination [38]. Whilst Maxwell interrupted his evolved programs at regular intervals and only those doing well were allowed to continue [30]. Other forms of GP allow other topologies or inclusion of memory [17], short cuts [5] or side effects [13].

Our model follows most GP systems and assumes that information flows steadily in a feed forward fashion from the outer most parts of the tree towards the root node (see left hand side of Figure 1 from [16, Koza video 11:10]).

1.3 Using Incremental Evaluation to Measure Failed Disruption Propagation

Last year we [19, 20] showed considerable savings can be obtained by replacing top down recursive evaluation of large GP trees by bottom up evaluation. Since there are no side effects the results are identical. Then we were mainly interested in speed and often found that similar trees had identical fitness. Here we explain this by showing in deep GP trees most disruptions fail to propagate as far as the tree's output.

Figure 1 shows a parent and one of its children which inherits its root node. Since the child was created by crossing over or mutating the parent, we can readily start bottom up evaluation from the point where they differ. By continuously comparing evaluations of parent and child, incremental evaluation [20] can easily see if they have become identical. Since the only code difference is below the crossover/mutation point, if at any function on the path between the code change and the root node their evaluations do become identical, they will remain identical. The right hand side of Figure 1 shows a small example in which incremental evaluation stops along the path from crossover point to root node at the % 4 levels deep because the % calculates an identical value in the parent and child. Therefore at this point it is known that both programs will return the same value. In a static environment, if they return the same values on all test cases, then they have identical fitness.

2 GP POPULATION CONVERGENCE

2.1 Genetic Programming Tree Shapes

As has been repeatedly reported [21], it is common for genetic programming without size or depth limits to generate trees of a similar shape to uniform random trees. That is, trees drawn at random from all the trees of a given size. The mathematics of such trees has been extensively studied [36] (see also Figures 2 and 3). Our model of genetic programming is thus based on uniformly sampling such trees and recent results from incremental evolution.

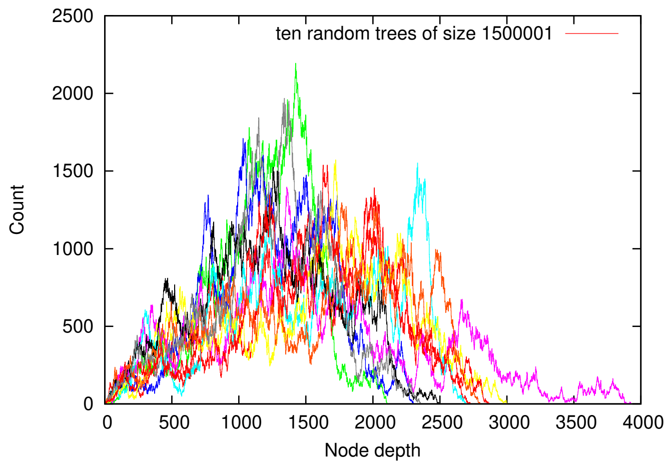

Figure 2: Distribution node depths in ten random trees of 1 500 001 nodes. (Mean for all trees of this size is 1534. See also Figure 3.)

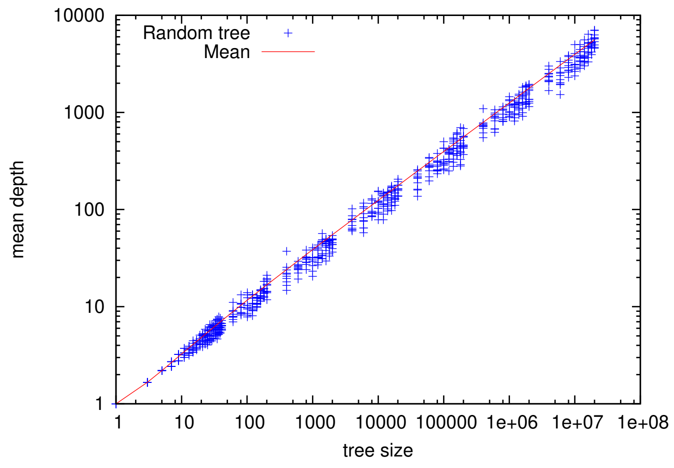

Figure 3: Mean distance to the root node for random trees of each size. (Typically it is about half the tree height.) The solid line shows the mean of the mean distances calculated for all trees of given size. For large trees it tends to $\approx \sqrt{\pi|\text{size}|/2}$. As the data + scatter shows, the mean distances vary widely. Indeed the coefficient of variation tends to about 18%. Note log scales.

2.2 Uniform Model of Tree Disruption Sites

For simplicity, we assume each point in a genetic programming tree is equally likely to be chosen for change[1]. The change can be made either by mutation or subtree crossover. We assume that, at the change site, the injected code gives a different value to that previously calculated by the original code[2].

[1] Koza [14] defines his crossover to have a 90% bias towards choosing internal functions for crossover points. However, particularly in large trees, this makes the calculations more complex and less general and, simply makes the path to the route node approximately one function shorter.

[2] Typically in GP the chance of replacing a randomly chosen subtree with another which gives the same value is at most only a few percent. For simplicity we assume it is zero percent.

2.3 Information Theory of Phenotypic Fitness Convergence

For simplicity, we start by considering Monte Carlo simulations (Section 3) with only a single test case. We assume that at each function between the disruption and the root node there is a fixed chance that the difference between the original and the new evaluation becomes zero.

Since the tree is hierarchical, once the difference between the evaluation of the two trees becomes zero, it must remain zero. To remain general, we do not model exactly how the difference becomes zero, we simply assume that there is some chance of the two evaluations synchronising.

Since each function is irreversible, it looses information about its inputs. (Figure 4 shows an example with two functions + and $*$ and three inputs x, y and z.) Once information is lost, it cannot be recovered. For example, in the case of 32-bit GP, values calculated by each function in the tree can contain at most 32 bits of information. In information theoretic terms [34], their entropy is at most 32 bits. In particular the conditional entropy of the function's output at the disruption point is at most 32 bits. But this falls monotonically with distance from the disruption. I.e., the distance above the changed code. Conversely, if the mother and child evaluations are to remain different, then at each function on the path from the disruption to the root node, the value calculated by that function must yield a different 32 bit pattern in mother and child. As the conditional entropy continually falls, the distribution of the frequencies of each the 32 bit patterns becomes less uniform, i.e. they bunch together. Thereby increasing the chance that the 32 bit patterns occupied by the evaluation of the old and new code, coincide. Once they do coincide, their evaluations are identical, and will remain identical, until the program terminates.

In GP systems, we are familiar with explicit mechanisms which make parent and offspring evaluations the same. For example, in a Boolean problem, we may encounter an AND function on the path to the root node whose other argument is false. In which case, no matter what the change to the GP tree, the evaluation of the AND will be false. That is, above the AND (including the output root node) the parent and offspring will have the same evaluation.

In a floating point GP problem, multiplication scales the difference between the parent and child's evaluations. A sequence of multiplications by values calculated by side subtrees, where the |values| are less than 1.0, perhaps mixed with linear operations (e.g. addition and subtraction), can rapidly reduce the magnitude of the difference. At some later point, a simple addition or subtraction by a value near 1.0 can cause rounding error to give the same value in both evaluations. (Section 6.2 contains 1533 such examples.) E.g., assume a sequence of n multiplies by values near 0.5 mixed with linear operations. Assume at the disruption point the original and changed code evaluations are old=1.5, new=2.5, so |diff|=1.0. Then n multiplications later $|\text{diff}|=1.0 \times 1/2^n$. For n>23 the difference becomes so small that rounding error on addition (or subtraction) of a value near 1.0, will give the same answer in both the old and new program. Similarly, if the values of the side subtrees are consistently outside ±1, multiplication will eventually lead to numeric overflow in both programs.

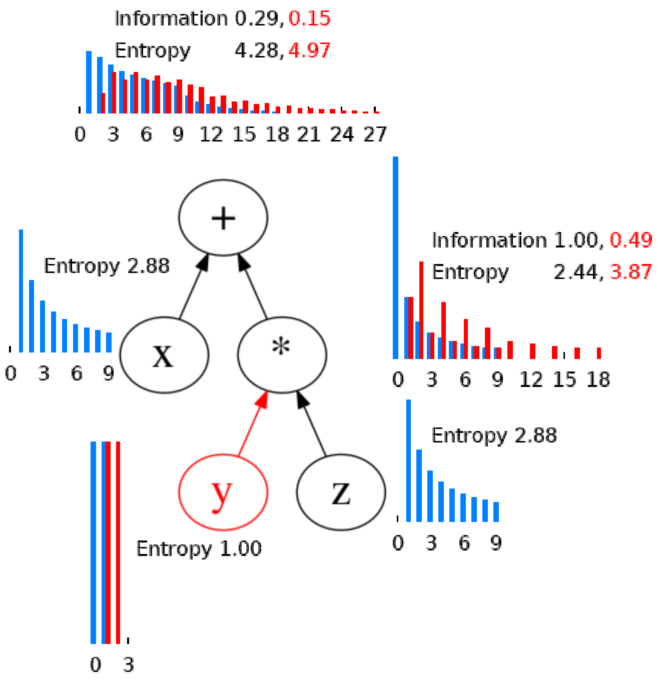

Figure 4: Information view of program (+ x (* y z)). Inputs x and z are independently drawn from a non-uniform distribution. The blue graphs show the distribution of evaluations at the five nodes in the program. Note y is originally either 0 or 1. The red graphs show the changed distributions after a mutation makes y +1 bigger (i.e. now 1 or 2). Before the mutation, at the multiplication node (*) it is easy to tell if y is zero or not (mutual information 1 bit). Adding x (top node) makes it harder, reducing mutual information to 0.29. (After the mutation, red, the mutual information is 0.49 and 0.15) Notice entropy is always ≤ sum of entropies of inputs.

2.4 Geometric Model of Convergence of GP Evaluations

To keep the model general, we do not need to consider the GP primitives and test case in detail. Instead we simply assume at each function on the path between the crossover point or mutation location to the root node there is a constant finite chance of the evaluation of the parent and child syncronising and once syncronsed they remain syncronised. The fixed chance p gives us a geometric probabilty distribution[3] (mean=$1/p$). For simplicity we deal with the characteristic length $1/p$. Earlier work, shows the characteristic length to be variable with type of GP and test case. Therefore we use a wide range of lengths (see next section). From 1.0 up to 130.0 (the maximum reported in GP). (Figure 12 confirms this range is reasonable.)

3 MONTE CARLO SIMULATIONS

The parameters are given in Table 1. We generate uniformly at random (binary) trees of all legal sizes up to 41 nodes, then increase tree size by 20 up to 201 then by 200 for trees up to size 2001, and so on. Thus we sample nine sizes for each order of magnitude. The largest samples have 20 000 001 nodes (74 sizes in total, see Table 1). As mentioned above (Section 2.2), we assume that every node in each tree (i.e., up to 20 000 001 points) can be disrupted, calculate the length of the path from it to the root node and then (using the geometric distribution) calculate the probability that the disruption will fail to propagate as far as the program's output (i.e., fail to reach the root node). For each node we repeat the calculation for five characteristic lengths (1, 10, 30, 70 and 130). For each tree size and each length we run ten experiments with different pseudo random number seeds. (Total 6 000 016 700 probabilities.)

[3] Figure 8 lower three lines, shows geometric probabilty distributions with p = 0.1, 0.2 and 0.3 (characteristic lengths $1/p$ = 10, 5, and 3.33).

Table 1: Monte Carlo Sampling Parameters for Failed Disruption Propagation (FDP) in Deep Trees

- Ten trees uniformly sampled from all those of given size [36],[4],[12],[18].
- Phenotypic change (either mutation or crossover) uniform across tree.
- Uniform chance of side subtree clearing phenotypic change, leading to geometric distribution of failed disruption propagation [32]
- Mean disruption lengths: 1, 10, 30, 70, 130
- Size of trees 1, 3 ... 41, 61 ... 201, 401 ... 2001, 4001 ... 20 001, 40 001 ... 200 001, 400 001 ... 2 000 001, 4 000 001 ... 20 000 001

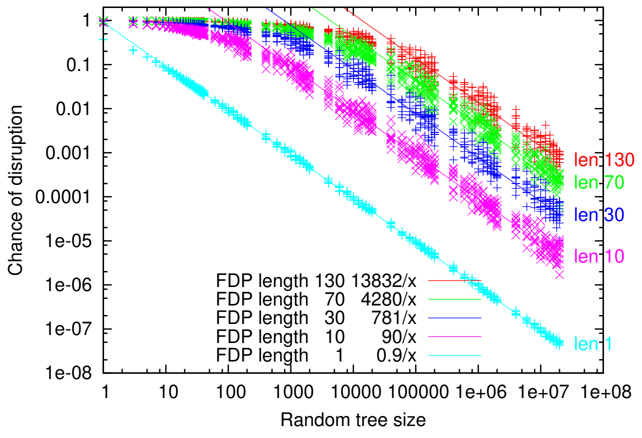

Figure 5: Predicted chance of crossover or mutation changing fitness for five geometric models and ten trees of each size. Straight lines are best RMS regression fits to a/x (slope -1). Note log scales.

The results are given in Figure 5. For every function and leaf in each tree the chance of crossover or mutation at that node changing the fitness of one test case is calculated for each of the five disruption lengths. Figure 5 plots the probability of any fitness change for that tree assuming each node is equally likely to be chosen for crossover or mutation.

As expected, Figure 5 shows for small trees almost all genetic changes disrupt fitness (probability ≈1). As we consider larger trees, the nodes are, on average, further from the root node (see Figure 3) and so, as expected, the chance of fitness changing falls to near zero for very large random trees.

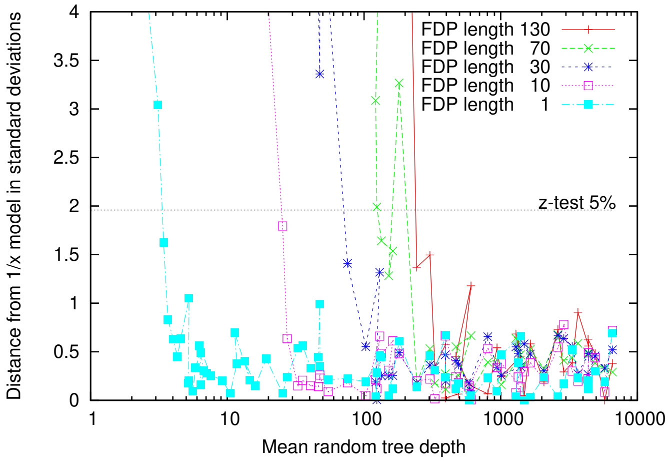

Figure 6: Difference between Monte Carlo measurements and $1/x$ model (Figure 5). Values below 1.96σ (dotted horizontal line) suggest a good fit for deep trees

If the chance of fitness disruption across the whole test suite is below 1/(the population size × the takeover time), then selection (e.g. tournament selection) will ensure almost every member of the population has the same fitness[4]. That is, the population will converge in terms of fitness (phenotypic convergence), even if each member of the population is different (no genetic convergence).

As a simple example, assume we have only one test case, a population of 500, and strong selection (take over time 4 generations). Thus even with 100% crossover, we expect phenotypic convergence when trees are deep enough to reduce the chance of fitness disruption below 1/2000. Figure 5 predicts in random trees, that this will depend very heavily on how effective our test case(s) is at passing disruption up the tree. For the least effective fitness test(s), the leftmost (light blue) line in Figure 5 predicts bloat to tree size 1800 (mean depth 52). But for FDP length 10 this rises to 180 000 (mean depth 531). (Length 30⟶1 562 000 (depth 1565), length 70⟶8 560 000 (depth 3666), and trees of size 27 664 000 (mean depth 6591) for a geometric model length of 130 functions.)

We also tried root mean squared (RMS) fitting of the log data with depth as well as size, using quadratic and linear models but a simple model assuming a fixed sized sensitive region a in trees of size x, which gives an a/x model works well. The regression lines in Figure 5 are the best RMS fit (in log space) for probability versus 1/size for each FDP length.

Figure 6 and Table 2 consider when the $1/x$ model applies by calculating the difference between its predictions and the Monte Carlo measurements. The difference is normalised by dividing by the observed standard deviation, and so is somewhat noisy. Nonetheless in each case Figure 6 makes clear the fit is very good (mostly within one standard deviation) once the tree depth exceeds a threshold. And the threshold depends upon how far the crossover disruption propagates (FDP length). See Table 2.

[4]Goldberg's takeover time is the expected number of generations taken by selection alone for all but one member of the population to have the same fitness [9].

Table 2: Smallest tree when reciprocal $1/x$ model holds. See also Figures 5 and 6. Columns described in Section 4.

FDP length	reciprocal model a/x	mean depth	mean depth/FDP	p	size	ratio size/a
1	0.9	3.2	3.23	4%	9.1	9.94
10	89.5	23.5	2.35	10%	308.1	3.44
30	780.9	66.7	2.22	11%	2871.7	3.68
70	4280.0	213.1	3.04	5%	21734.4	5.08
130	13832.3	348.0	2.68	7%	36703.6	2.65

4 SIZE OF SENSITIVE AREA BY ROOT NODE

Table 2 lists for each of our five geometric FDP lengths: the numeric value of the best RMS fit shown in Figure 5, and the minimum depth and size of trees to which the RMS fit applies. I.e. the shallowest tree when the model prediction's is less than three standard deviations out. (Columns 3 and 6 are calculated using linear interpolation.) We can interpret the results as telling us that, for each FDP length, there is a sensitive area around the root node. If we consider bigger trees, the chance of a crossover landing in the sensitive area falls directly in proportion to the size of the tree. Hence the $1/x$ law found in Figure 5. That is, the size of the sensitive region is fixed. The second column in Table 2 tells us its size. The fourth column expresses the mean depth (col 3) as a multiple of characteristic length for our geometric model (col 1). Notice col 4 tells us that once the average tree depth is about 2 or 3 times the characteristic depth, the $1/x$ fit applies. That is, crossover or mutations deeper than 2–3 times the characteristic depth have little chance of changing fitness. (The fifth column, p, holds the chance as a percentage % of disruption given by the geometric model for a change at the mean depth, col 3.)

Empirically the size of sensitive region near the root is about the same size as an average tree of height 2.37 × the mean FDP length. The last column in Table 2 confirms, excluding the very shortest FDP length (1), the $1/x$ rule is highly predictive for trees that are more than about three times this size.

5 LIMITED EFFECTIVENESS OF MANY TESTS

5.1 n Independent Test Cases with the Same Propagation give $\log n$ Impact

Figure 7 shows that our geometric model predicts that if each member of the GP test set is equally effective and independent then the effectiveness of the whole test suite (of n tests) grows only slowly (as $\log n$) as the number of tests is increased. (We hinted at $\log n$ dependence for independent tests in [21, 24, 25] but closely spaced tests may not be independent, making additional tests even less effective.)

Recall our failed disruption propagation model assumes that at each function from the genetic change to the root node there is a small chance p of the evaluation (for a given test case) of the parent and offspring synchronising. This gives rise to a geometric distribution, $p(1-p)^{k-1}$. Define len $= 1/p$. len is the mean of the geometric distribution. Thus if the crossover or mutation point is many times len from the root node, there is essentially no chance it will change the child's overall evaluation. To be definite, if we want

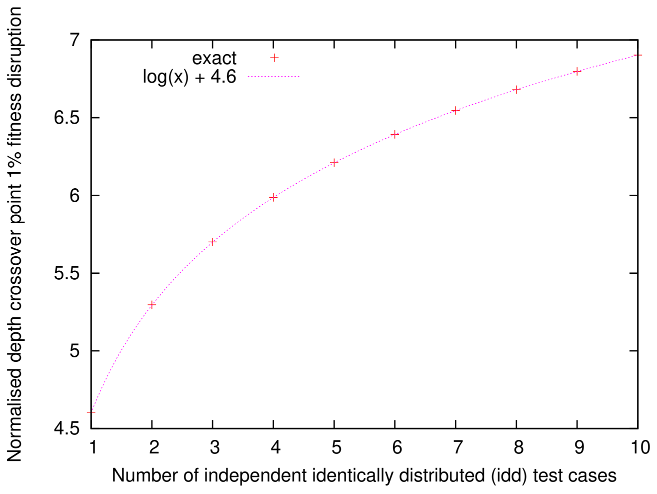

Figure 7: Number of functions crossover or mutation phenotypic disruption must pass through before reaching the root node before the chance of changing fitness is less than 1% versus test suite size. + calculated for large mean (small p). Vertical axis normalised by dividing by mean of geometric distribution.

to be 99% sure the child's evaluation is the same as its parent then:

$$\begin{aligned} 1-(1-p)^k &\geq 0.99 \\ (1-p)^k &\leq 0.01 \\ k\log(1-p) &\leq \log(0.01) \\ k &\geq \log(0.01)/\log(1-p) \\ k &\geq -\frac{1}{p}\log(0.01) = \frac{1}{p}\log(100) = \frac{4.60517}{p} = 4.60517 \text{ len} \end{aligned}$$

That is, to be 99% sure there is no change in evaluation on one test case, the crossover has to be deeper than 4.6 times len. (We have used $p \ll 1$ so $\log(1-p) \approx -1/p$ is reasonably tight.) If we increase the number of tests n this depth increases only slowly.

Suppose we have n independent tests. If none cause an evaluation change, then there is no fitness change. We calculate the depth to be 99% sure of this. We need all n tests to be give the same answers they gave before, so

$$\begin{aligned} \left(1-(1-p)^k\right)^n &\geq 0.99 \\ 1-(1-p)^k &\geq \sqrt[n]{0.99} \\ k\log(1-p) &\leq \log\left(1-\sqrt[n]{0.99}\right) \\ k &\geq \log\left(1-\sqrt[n]{0.99}\right)/\log(1-p) \\ k &\geq -\frac{1}{p}\log\left(1-\sqrt[n]{0.99}\right) \\ k &\geq -\frac{1}{p}\log\left(1-\exp\left(\frac{1}{n}\log(0.99)\right)\right) \\ k &\gtrsim -\frac{1}{p}\log\left(\frac{0.0100503}{n}\right) \\ k &\gtrsim \frac{1}{p}(\log(n)+4.60015) \qquad (1) \end{aligned}$$

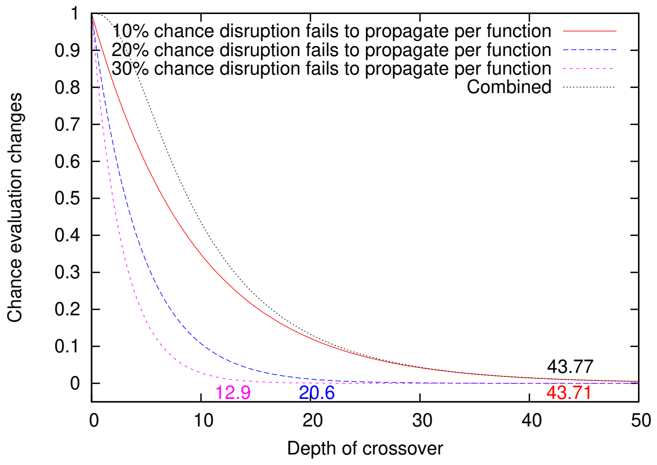

Figure 8: Probability of fitness change for three independent tests assuming geometric model of failed disruption propagation (FDP) with different abilities to measure genetic changes. Dotted line shows probability of any fitness change when all used. Numbers above horizontal axis give depth where chance of evaluation changing is 0.01. Note at depth, less effective tests (20%, 30%) add little.

[5] Notice dependence on log n. (See also Figure 7.)

In continuous GP problems, e.g. floating point, test values which are close together may not be independent but instead may behave similarly. That is, they may have similar abilities to detect changes within the GP tree. Therefore, from an efficiency point of view, they add little to the test set. Where fitness tests have similar abilities to propagate disruption up GP trees, we expect Equation 1 (and Figure 7) to apply, if we treat n as the number of independent tests, rather than the total size of the test set.

5.2 Independent Test Cases with Different Propagation

Of course if tests are not independent, then increasing the number of them may not help at all. Also, even if tests are independent, adding weaker ones to the test suite makes little difference.

Figure 8 shows the combined effectiveness of three independent tests, each with a geometric distribution. In this example, the mean lengths are 10 (1/10=10%), 5 (1/5=20%) and 3.33 (1/3.33=30%). Notice the weaker tests (20% dashed blue) and (30% dashed purple) make little difference to the combined (black dotted) fitness function's ability to measure the impact of deep genetic changes.

6 FAILED DISRUPTION PROPAGATION IN DEEP FIT GP TREES

In the final experiments (Sections 6.1 and 6.2) we demonstrate the robustness of deep evolved GP trees by changing the evaluation of the trees at every location in the tree and tracing how far this artificial disruption travels before either reaching the root node or being quenched by FDP within the tree.

[5]Note $\log(0.99)$ = -0.0100503 and $\log(-\log(0.99)) = -4.60015$

Table 3: Failed Disruption Propagation in Deep GP Trees

Terminal set:	X, 250 constants -0.995 to 0.997
Function set:	MUL ADD DIV SUB (Section 6.1)
	MUL ADD (Section 6.2)
Fitness cases:	48 fixed input -0.97789 to 0.979541 (randomly selected in -1.0 to +1.0). Target Sextic polynomial $y = xx(x-1)(x-1)(x+1)(x+1)$
Selection:	Fitness = $\frac{1}{48}\sum_{i=1}^{48}\lvert GP(x_i) - y_i\rvert$ tournament size 7
Population:	Panmictic, non-elitist, generational.
GP parameters:	Initial population 500 ramped half and half [14] depth between 2 and 6. 100% unbiased subtree crossover. 600 generations. No size or depth limit.

DIV and % denote protected division `(y!=0)? x/y : 1`

In both Section 6.1 and 6.2 we run GP ten times with a population of 500 for 600 generations on Koza's Sextic polynomial problem [14] with 48 test cases (see Table 3) before choosing a high fitness tree and reevaluating it on all 48 test cases[6] on each node within it but with the evaluation increased by 1.0 [7]. Note we do not change the tree's genetic material, but simply inject this large perturbation into every point in its fitness evaluation[7]. Naturally disruptions near the root node often change its fitness but, as we shall see, those deep within the tree often do not.

In Section 6.1 we use the traditional four GP functions (addition, multiply, subtraction and protected division). On average 83.3% of disruptions fail to propagate as far as the root node. A small fraction of these are halted simultaneously on all 48 test cases by a "classic intron" such as multiplication or division by zero, which produce the same result regardless of the function's other argument (see end of Section 6.1). In Section 6.2 we run GP without subtraction and protected division. This dramatically reduces the chance of the constant zero being evolved. (In fact, it does not occur at all in our examples.) But we still see failed disruption propagation.

6.1 GP Symbolic Regression

After 600 generations the evolved trees are deep enough so that in most cases the injected disruption fails to propagate to the root node. Table 4 shows if the tree depth exceeds 200, more than 80% of our changes do not impact the values calculated by the GP tree on the fitness cases. It gives the tree depth and corresponding mean fraction of the tree where our disruption of the evaluation did make a difference at the root node, averaged over all 48 test cases. As expected, deeper trees tend to have lower disruption. Figures 9 and 10 show that with fitness test values near zero, the disruption tends to propagate up through a smaller number of functions in the GP tree, and therefore they are less good at detecting the changed evaluations.

Figure 9 shows failed disruption propagation by test case for ten highly evolved deep fit trees (fitness 6.4 10^{-5}–0.036, size 7415–546 441, depth 126–2794). Figure 9 does not plot the 16.7% of cases where the perturbation did reach the root. Note initial rapid rise in fraction of evaluations where disruption is lost, followed by approximately log-linear tails.

Table 4: Run time perturbation of ten evolved GP trees

Run	Tree size	Depth	Fitness changed
1	32833	350	12.2%
2	546441	2794	0.2%
3	31925	363	17.1%
4	50679	300	19.6%
5	27147	174	38.6%
6	24437	649	7.2%
7	28105	388	4.0%
8	16747	273	19.8%
9	7415	126	27.1%
10	10685	175	21.4%

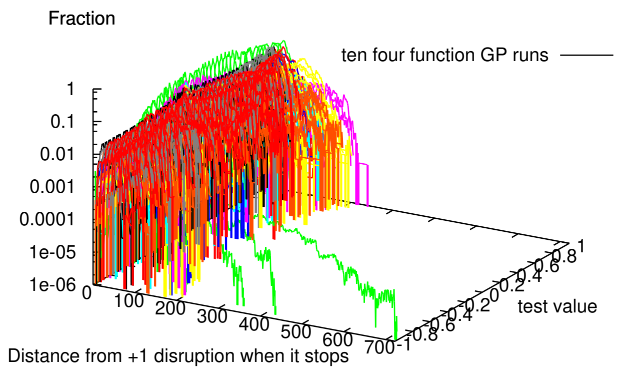

Figure 9: Distance disruption caused by adding 1.0 travels before evaluation on test case becomes identical. Ten GP Sextic runs for 600 generations. Log vertical scale.

On average 83.3% of perturbations do not reach the root node. Most (72.8%) do not stop at the same point for all 48 test cases. However some cases of disruption failing to propagate are classic introns. Across the ten runs there are a total of 23 952 cases (10.5% of perturbations) where all 48 test cases fail to propagate at the same point in the evaluation of the tree. 23 481 are when either multiplication by zero or division by zero yields 0 or 1, on all test cases regardless of their other argument (i.e. classic introns). The remaining 471 (0.4%) are on addition or subtraction where one input is very small but is combined (by + or −) with a value near 1.0, so that rounding error means the result is the same as it was before the injected perturbation.

6.2 Robust Evolved GP Polynomials

We repeated the experiments in Section 6.1 again ten times but without subtraction or division (see Table 3). I.e. GP evolved polynomials. (Across the ten polynomial examples fitness, i.e. average |error|, 0.033 to 5.0 10^{-5}, size 8827 to 863 131 and depth 121 to 5103.) Omitting subtraction and division prevented the evolution of nodes which evaluated to zero for all test cases and thus there are no classic introns. Indeed in the ten runs there are only 1533 (0.08%) cases (excluding the root nodes) where the evaluation perturbation is lost by all test cases at the same point. (All are due to rounding

[6]Notice we are re-using the testset, so if the trees have overfit we are testing for FDP exactly where they might be expected to do well. Also Section 5 has shown at best increasing the number of tests n above 48 only makes a logarithmic difference ($\log(n)$).
[7]We are studying run time perturbation. It can also be viewed as a somewhat realistic approximation to genetic disruption since typically GP genetic operations, such as crossover and mutation, choose at random from all parts of the parent tree and in the Sextic polynomial changes in evaluation at the point where the code has been changed are of the order of 1.0.

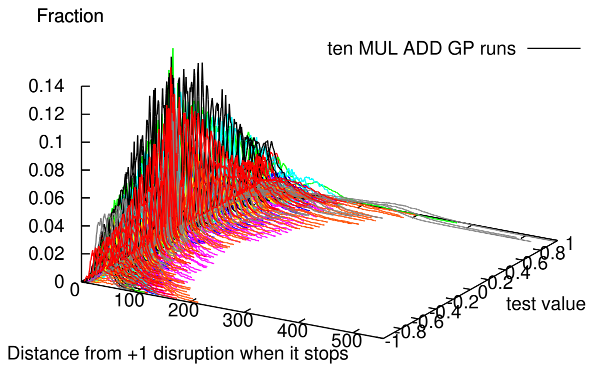

Figure 10: Distance +1.0 disruption travels before evaluation on test case becomes identical to the unperturbed evaluation. Ten GP runs 600 generations. (Linear scales.)

error on addition.) Figure 10 shows how far disruption propagated in ten deep fit evolved polynomials by test case.

Figure 11 shows in detail one test case from Figure 10. (We chose the smallest test value, which is also the central test case, number 25, X=0.00546999). Depending upon the run, the maximum of the distribution is between 4 and 35 functions above the point in the tree where the evaluation was initially disrupted. However it then falls approximately log-linearly as predicted by a geometric distribution. That is, the geometric distribution is only partially correct and some "warm up" is needed before we reach the peak but it models the fall afterwards (Figures 9 and 10).

Figure 11 shows the log-linear RMS best fit to the right of the peak in the curves in Figure 11 for fitness test case 25 on each of the ten runs. (These best fits are shown as straight lines in Figure 11.) To avoid excessive noise, the best RMS fit ignores data with less than ten instances. Figures 11 and 12 give the slope as the mean number of functions traversed (assuming the continuous, exponential, distribution started at 1.0).

Notice in keeping with the very different GP trees evolved in the ten runs, there is considerable variation between the 10 runs shown in Figures 10 to 12. However Figure 12 shows a trend for disruption to fail to propagate more quickly with GP fitness test values that are near zero than for values near -1 or +1.

7 CONCLUSION

Information loss is inevitable in digital computing. We have argued that it causes failed disruption propagation (FDP). We show FDP is common in deep genetic programming trees, and in many cases run time perturbations of similar size to those caused by crossover and mutation are invisible to fitness test cases. If all tests fail to detect genetic change, offspring inherit identical fitness and selection then causes the population to phenotypically converge despite genetic diversity and useful evolution stops.

We have presented a geometric model of failed disruption propagation and shown it has some similarity to deep evolved GP trees. Our mathematical model (Equation 1) quantifies the slow logarithmic increase in the effectiveness of test suites with number n of equally effective independent tests. Notice, that without additional, white box, insight, $O(\log n)$ is the best that can be done. In contrast:

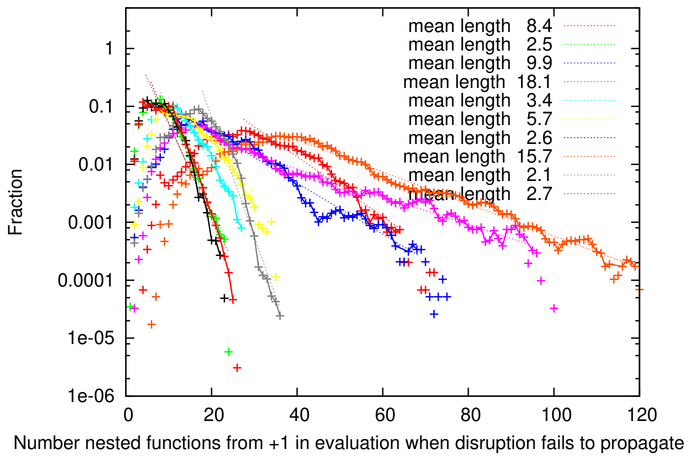

Figure 11: Straight lines show the log-linear (geometric) fall with distance that disruption caused by adding 1.0 travels before evaluation on middle test case (25) becomes identical to unperturbed evaluation. Ten MUL ADD GP runs. As Figure 10, excludes small number of cases which did reach root or where all test cases behave the same. Log vertical scale.

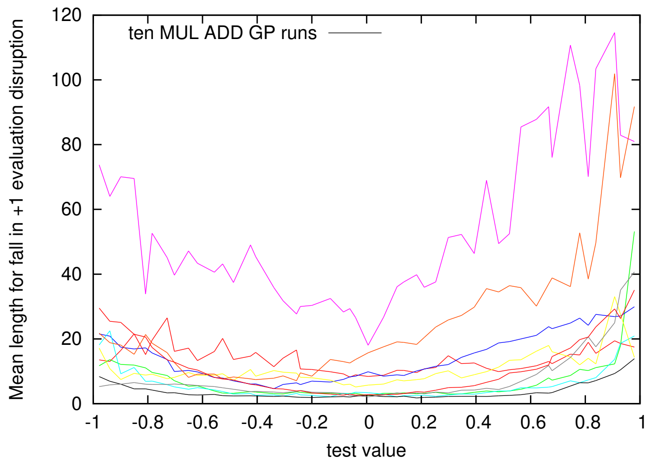

Figure 12: RMS fit assuming geometric model to +1 disruption data in Figure 10 for each fitness test value. Note despite variation between evolved GP trees, disruption of the evaluation with test values near zero (centre of graph) is more rapidly quenched than for values near ±1. Data for test case 25 (center) plotted in detail in Figure 11.

1) Similar test values may not be independent and so add little to the test suite as a whole. 2) The effectiveness of fitness test sets composed of tests of mixed abilities are dominated by the most effective and the weak ones (e.g. perhaps values near zero) add little.

Acknowledgements

I am grateful for the assistance of the anonymous reviewers. Funded by Meta OOPS project.

REFERENCES

[1] Rawad Abou Assi, Chadi Trad, Marwan Maalouf, and Wes Masri. 2019. Coincidental correctness in the Defects4J benchmark. *Software Testing, Verification and Reliability* 29, 3 (2019), e1696. http://dx.doi.org/10.1002/stvr.1696

[2] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. 1998. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications.* Morgan Kaufmann, San Francisco, CA, USA. https://www.amazon.co.uk/Genetic-Programming-Introduction-Artificial-Intelligence/dp/155860510X

[3] Ali Basiri, Niosha Behnam, Ruud de Rooij, Lorin Hochstein, Luke Kosewski, Justin Reynolds, and Casey Rosenthal. 2016. Chaos Engineering. *IEEE Software* 33, 3 (May-June 2016), 35–41. http://dx.doi.org/10.1109/MS.2016.60

[4] Walter Bohm and Andreas Geyer-Schulz. 1996. Exact Uniform Initialization for Genetic Programming. In *Foundations of Genetic Algorithms IV*, Richard K. Belew and Michael Vose (Eds.). Morgan Kaufmann, University of San Diego, CA, USA, 379–407. http://cseweb.ucsd.edu/~rik/foga4/Abstracts/07-wb-abs.html

[5] Alexander Brownlee, Justyna Petke, and Anna F. Rasburn. 2020. Injecting Shortcuts for Faster Running Java Code. In *2020 IEEE Congress on Evolutionary Computation (CEC)*, Alexander (Sandy) Brownlee, Saemundur O. Haraldsson, Justyna Petke, and John R. Woodward (Eds.). IEEE, Internet. http://dx.doi.org/10.1109/CEC48606.2020.9185708 Special Session on Genetic Improvement.

[6] Vic Ciesielski, Dylan Mawhinney, and Peter Wilson. 2002. Genetic Programming for Robot Soccer. In *RoboCup 2001: Robot Soccer World Cup V (Lecture Notes in Computer Science, Vol. 2377)*, Andreas Birk, Silvia Coradeschi, and Satoshi Tadokoro (Eds.). Springer, Seattle, Washington, USA, 319–324. http://dx.doi.org/10.1007/3-540-45603-1_37

[7] Benjamin Danglot, Philippe Preux, Benoit Baudry, and Martin Monperrus. 2018. Correctness attraction: a study of stability of software behavior under runtime perturbation. *Empirical Software Engineering* 23, 4 (1 Aug. 2018), 2086–2119. http://dx.doi.org/10.1007/s10664-017-9571-8

[8] Charles Darwin. 1859. *The Origin of Species* (penguin classics, 1985 ed.). John Murray.

[9] David E. Goldberg and Kalyanmoy Deb. 1991. A Comparative Analysis of Selection Schemes Used in Genetic Algorithms. In *Proceedings of the First Workshop on Foundations of Genetic Algorithms*, Gregory J. E. Rawlins (Ed.). Morgan Kaufmann, San Mateo, 69–93. http://dx.doi.org/10.1016/B978-0-08-050684-5.50008-2

[10] Mark Harman, Yue Jia, and William B. Langdon. 2010. A Manifesto for Higher Order Mutation Testing. In *Mutation 2010*, Lydie du Bousquet, Jeremy Bradbury, and Gordon Fraser (Eds.). IEEE Computer Society, Paris, 80–89. http://dx.doi.org/10.1109/ICSTW.2010.13 Keynote.

[11] Ting Hu, Marco Tomassini, and Wolfgang Banzhaf. 2020. A network perspective on genotype-phenotype mapping in genetic programming. *Genetic Programming and Evolvable Machines* 21, 3 (Sept. 2020), 375–397. http://dx.doi.org/10.1007/s10710-020-09379-0 Special Issue: Highlights of Genetic Programming 2019 Events.

[12] Hitoshi Iba. 1996. Random Tree Generation for Genetic Programming. In *Parallel Problem Solving from Nature IV, Proceedings of the International Conference on Evolutionary Computation (LNCS, Vol. 1141)*, Hans-Michael Voigt, Werner Ebeling, Ingo Rechenberg, and Hans-Paul Schwefel (Eds.). Springer Verlag, Berlin, Germany, 144–153. http://dx.doi.org/10.1007/3-540-61723-X_978

[13] David Jackson. 2012. Single Node Genetic Programming on Problems with Side Effects. In *Parallel Problem Solving from Nature, PPSN XII (part 1) (Lecture Notes in Computer Science, Vol. 7491)*, Carlos A. Coello Coello, Vincenzo Cutello, Kalyanmoy Deb, Stephanie Forrest, Giuseppe Nicosia, and Mario Pavone (Eds.). Springer, Taormina, Italy, 327–336. http://dx.doi.org/10.1007/978-3-642-32937-1_33

[14] John R. Koza. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA. http://mitpress.mit.edu/books/genetic-programming

[15] John R. Koza. 1994. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press, Cambridge Massachusetts. http://www.genetic-programming.org/gpbook2toc.html

[16] John R. Koza and James P. Rice. 1992. *Genetic Programming: The Movie.* MIT Press, Cambridge, MA, USA. https://youtu.be/tTMpKrKkYXo

[17] William B. Langdon. 1998. *Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming!* Genetic Programming, Vol. 1. Kluwer, Boston. http://dx.doi.org/10.1007/978-1-4615-5731-9

[18] William B. Langdon. 2020. *Fast Generation of Big Random Binary Trees.* Technical Report RN/20/01. Computer Science, University College, London, Gower Street, London, UK. https://arxiv.org/abs/2001.04505

[19] William B. Langdon. 2021. Fitness First. In *Genetic Programming Theory and Practice XVIII (Genetic and Evolutionary Computation)*, Wolfgang Banzhaf, Leonardo Trujillo, Stephan Winkler, and Bill Worzel (Eds.). Springer, East Lansing, MI, USA, 143–164. http://dx.doi.org/10.1007/978-981-16-8113-4_8

[20] William B. Langdon. 2021. Incremental Evaluation in Genetic Programming. In *EuroGP 2021: Proceedings of the 24th European Conference on Genetic Programming (LNCS, Vol. 12691)*, Ting Hu, Nuno Lourenco, and Eric Medvet (Eds.). Springer Verlag, Virtual Event, 229–246. http://dx.doi.org/10.1007/978-3-030-72812-0_15

[21] W. B. Langdon. 2022. Genetic Programming Convergence. *Genetic Programming and Evolvable Machines* 23, 1 (March 2022), 71–104. http://dx.doi.org/10.1007/s10710-021-09405-9

[22] W. B. Langdon and B. F. Buxton. 2001. Genetic Programming for Improved Receiver Operating Characteristics. In *Second International Conference on Multiple Classifier System (LNCS, Vol. 2096)*, Josef Kittler and Fabio Roli (Eds.). Springer Verlag, Cambridge, 68–77. http://dx.doi.org/10.1007/3-540-48219-9_7

[23] William B. Langdon and Justyna Petke. 2015. Software is Not Fragile. In *Complex Systems Digital Campus E-conference, CS-DC'15 (Proceedings in Complexity)*, Pierre Parrend, Paul Bourgine, and Pierre Collet (Eds.). Springer, 203–211. http://dx.doi.org/10.1007/978-3-319-45901-1_24 Invited talk.

[24] William B. Langdon, Justyna Petke, and David Clark. 2021. Dissipative Polynomials. In *5th Workshop on Landscape-Aware Heuristic Search (GECCO 2021 Companion)*, Nadarajen Veerapen, Katherine Malan, Arnaud Liefooghe, Sebastien Verel, and Gabriela Ochoa (Eds.). ACM, Internet, 1683–1691. http://dx.doi.org/10.1145/3449726.3463147

[25] William B. Langdon, Justyna Petke, and David Clark. 2021. Information Loss Leads to Robustness. IEEE Software Blog. http://blog.ieeesoftware.org/2021/09/information-loss-leads-to-robustness-w.html

[26] W. B. Langdon and R. Poli. 1998. Why Ants are Hard. In *Genetic Programming 1998: Proceedings of the Third Annual Conference*, John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo (Eds.). Morgan Kaufmann, University of Wisconsin, Madison, Wisconsin, USA, 193–201. http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/WBL.antspace_gp98.pdf

[27] W. B. Langdon and Riccardo Poli. 2002. *Foundations of Genetic Programming.* Springer-Verlag. http://dx.doi.org/10.1007/978-3-662-04726-2

[28] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *Commun. ACM* 62, 12 (Dec. 2019), 56–65. http://dx.doi.org/10.1145/3318162

[29] Sean Luke, Charles Hohn, Jonathan Farris, Gary Jackson, and James Hendler. 1997. Co-evolving Soccer Softbot Team Coordination with Genetic Programming. In *Proceedings of the First International Workshop on RoboCup, at the International Joint Conference on Artificial Intelligence.* Nagoya, Japan. http://www.cs.gmu.edu/~sean/papers/robocupc.pdf

[30] Sidney R. Maxwell III. 1994. Experiments with a coroutine execution model for genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, Vol. 1. IEEE Press, Orlando, Florida, USA, 413–417a. http://dx.doi.org/10.1109/ICEC.1994.349915

[31] Martin Monperrus. 2017. Principles of Antifragile Software. In *Companion to the First International Conference on the Art, Science and Engineering of Programming* (Brussels, Belgium) *(Programming '17)*. ACM, New York, NY, USA, Article 32, 4 pages. http://dx.doi.org/10.1145/3079368.3079412

[32] Justyna Petke, David Clark, and William B. Langdon. 2021. Software Robustness: A Survey, a Theory, and Some Prospects. In *ESEC/FSE 2021, Ideas, Visions and Reflections*, Paris Avgeriou and Dongmei Zhang (Eds.). ACM, Athens, Greece, 1475–1478. http://dx.doi.org/10.1145/3468264.3473133

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

[34] Alfred Renyi. 1987. *A Diary on Information Theory.* John Wiley and Sons, Chichester.

[35] Eric Schulte, Zachary P. Fry, Ethan Fast, Westley Weimer, and Stephanie Forrest. 2014. Software mutational robustness. *Genetic Programming and Evolvable Machines* 15, 3 (Sept. 2014), 281–312. http://dx.doi.org/10.1007/s10710-013-9195-8

[36] Robert Sedgewick and Philippe Flajolet. 1996. *An Introduction to the Analysis of Algorithms.* Addison-Wesley.

[37] Ali Shahrokni and Robert Feldt. 2013. A Systematic Review of Software Robustness. *Information Systems and Technology* 55, 1 (January 2013), 1–17. http://dx.doi.org/10.1016/j.infsof.2012.06.002

[38] Astro Teller. 1994. Genetic Programming, Indexed memory, the Halting problem, and other curiosities. In *Proceedings of the 7th annual Florida Artificial Intelligence Research Symposium.* IEEE Press, Pensacola, Florida, USA, 270–274. http://www.cs.cmu.edu/afs/cs/usr/astro/public/papers/Curiosities.ps