Deep Genetic Programming Trees are Robust

WILLIAM B. LANGDON, Department of Computer Science, University College London, UK

We sample the genetic programming tree search space and show it is smooth since many mutations on many test cases have little or no fitness impact. We generate uniformly at random high order polynomials composed of 12 500 and 750 000 additions and multiplications and follow the impact of small changes to them. From information theory 32 bit floating point arithmetic is dissipative and even with 1501 test cases deep mutations seldom have any impact on fitness. Absolute difference between parent and child evaluation can grow as well as fall further from the code change location but the number of disrupted fitness tests falls monotonically. In many cases deeply nested expressions are robust to crossover syntax changes, bugs, errors, run time glitches, perturbations, etc., because their disruption falls to zero, and so it fails to propagate beyond the program.

Additional Key Words and Phrases: heritability, information theory, information funnels, sandpile 1/f powerlaw, self-organised criticality, SOC, self-similar fractal, GP fitness landscape, evolvability, mutational robustness, neutral networks, SBSE, software robustness, correctness attraction, diversity, software testing, theory of bloat, introns, error hiding, invisible faults, failed disruption propagation, FDP, FEP

ACM Reference Format:
William B. Langdon. 2022. Deep Genetic Programming Trees are Robust. *ACM Transactions on Evolutionary Learning and Optimization,* 2, 2, Article 6 (August 2022), 35 pages. https://doi.org/10.1145/3539738

1 INTRODUCTION

In software engineering there is increasing realisation that programs are not as fragile as is commonly assumed [29], [41],[43], [38], [58], [7], [14]. Indeed there is increasing realisation that many errors in either the source code or at run time have to propagate through the executing system to a point (e.g. a print statement) where their effect is visible, before the error is of concern. An analogy with civil engineering: concrete often contains microscopic cracks. Rather than trying to prevent them, engineers typically load the structure so that these cracks do not grow large enough to be important. However, unlike concrete, even *en-masse* it is difficult to model software. We use information theory to argue that in many cases data corrupted by an error or bug has to pass though dissipative code[1] before it reaches a point where it changes the program's outputs and in many cases the dissipative code loses the impact of the error, effectively rendering the error invisible [42].

[1] In thermodynamics a classic example of a dissipative process is a tornado. It is far from equilibrium and although violently mixing, the atmosphere containing it does not immediately become uniform. By dissipative code, we mean code which churns up the ordered information it takes in. Thus, while the disruption caused by an error may be readily discerned in the input data, after passing through dissipative code, the impact of the error is no longer obvious.

Author's address: William B. Langdon, W.Langdon@cs.ucl.ac.uk, Department of Computer Science, University College London, Gower Street, London, , UK, WC1E 6BT, UK.

2688-3007/2022/8-ART6
https://doi.org/10.1145/3539738

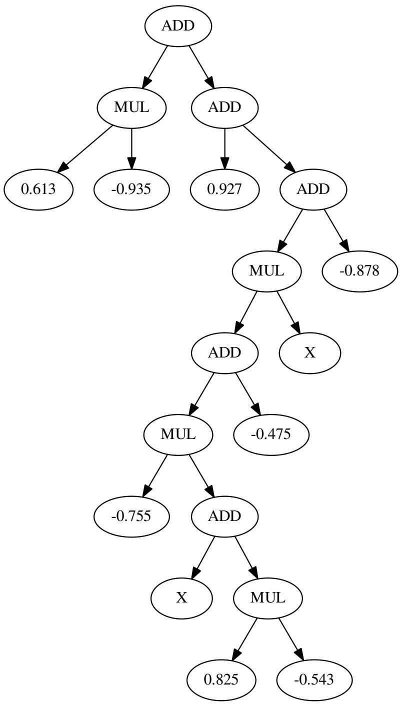

Fig. 1. Example subexpression $(0.613 \times -0.935) + 0.927 + (-0.755 \times (x + (0.825 \times -0.543)) - 0.475)x - 0.878$ as a tree. The value of the expression is given by the root node, here ADD, at the top [30].

We shall argue that computer programs are composed of irreversible operations which must lose information and are therefore dissipative[2]. For example, addition is irreversible[3]. We then concentrate upon floating point arithmetic operations. We use + and × to build genetic programming (GP) trees[4]. We show that even something as simple × + expressions, if deep enough, are robust. It is impractical to consider experiments on every GP function set. By demonstrating genetic disturbance often has no impact in what we argue is the simplest floating point case we hope to make the general argument persuasive. Indeed, we suggest that the greater information loss possible with other function sets, may lead to faster loss of disruption and so make polynomials an upper bound on disruption propagation. We inject errors (mutations) into these trees and then trace their impact. In many cases, particularly for deep errors, their impact is dissipated before it can reach the tree's output at its root node. That is, we use random sampling of the GP search space to show that information hiding makes most of the GP fitness landscape smooth.

Our approach is empirical. In Section 3 we describe how we uniformly sample from the genetic programming space. Then the experimental Section (Section 4) tracks the dissipation of the effect of tens of thousands of representative code changes made hundreds or even thousands of levels

[2]Due to being irreversible, all computation leads to information loss (which can be measured in terms of entropy). Therefore dissipative regions of programs have high entropy loss.

[3]Suppose `x=a+b;` and `x` is 1.5, then `a` could be 1.0 and `b` 0.5, or `a` could be -1.0 and `b` 2.5, etc., etc. That is, we cannot infer the values of `a` and `b` from just the output of `+`, and so `+` cannot be reversed.

[4] Figure 1 shows an example polynomial as a tree, whilst Figure 2 shows a bigger random expression again as a tree.

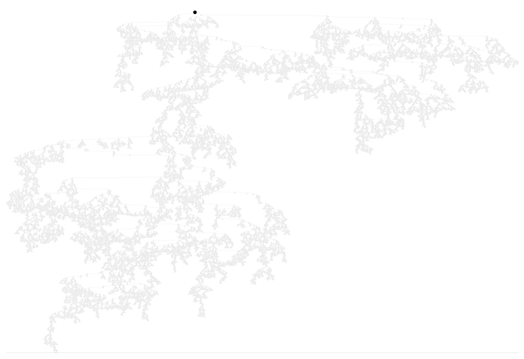

Fig. 2. A random polynomial (fun 0) presented as a binary tree of 25 001 nodes (depth 383). Root node (black circle) at top [30].

away from the program's exit point. In the discussion: Sections 5.1 to 5.4 develop empirical models to explain the mechanisms behind the observations. Whilst Sections 5.5 and 5.7 describe a few implications for genetic programming and software engineering. After we conclude in Section 6, there is an appendix (A) with definitions of some terminology. First in the next section, we show there is little existing work on GP trees thousands of levels deep, and then describe that they are of interest, since their analysis shows that search spaces for GP and programming in general, can be smoother than is often assumed.

2 BACKGROUND AND MOTIVATION

2.1 Background: Literature on Deep Genetic Programming Trees

In Section 4 we will present results for trees with an average depth of 400 or even 3000. This is in stark contrast to traditional genetic programming, which follows Koza's example, who in the first GP book [19] deliberately limited the evolved tree's depth to 17 levels. Although more restrictive size or depth limits and more sophisticated bloat control methods are also common, the 17 levels limit has been wide spread ever since [46]. Even Bi et al. [4, Sect.3.3] limit their image classification deep forest to depth "10–100" (although some of their evolved solutions have a depth of only eight [4, Fig. 6]), whilst Smith and Heywood [53] test their deep hierarchies of teams up to level eight.

2.2 Background Bloat: Why Some Changes are Invisible

In genetic programming [19, 46] the idea of useless bloated code is well known. Indeed by a somewhat imperfect analogy with mRNA transcription in biology, the name "intron" [55],[2],[49] is often used to refer to a part of a GP tree that has no impact on its output. For example, a subtree "ANDed" with false has no effect as the AND node will always evaluate to false regardless of the contents of the intron. Similarly multiplying by 0 always gives 0, meaning the other subtree has no effect and so is an intron. As the intron subtree has no impact on the tree's output it has no impact on its fitness.

2.3 Background: Evolution With Fitness Invariant Changes

Notice that intron behaviour is *heritable* [39, 47]. Although defined in terms of fitness evaluation (i.e. run time or phenotype), a child can inherit an intron subtree. If the other subtree of the function calling it is not disrupted by mutation or crossover, then the intron subtree remains an intron. That is, the inherited subtree, which had no impact on the parent's fitness, also has no impact on the child's fitness. For example, if the parent contains `(× intron zero-subtree)`, where `zero-subtree` is code that evaluates to zero, and the child inherits it, then the `intron` subtree is still an intron in the child. Further, even if the `intron` subtree is mutated or changed by crossover, the modified `intron'` subtree still has no impact on the child's fitness. For example, if `intron` was `(+ X 0.997)` and is mutated into `intron' = (+ X 0.93)` then `(× (+ X 0.93) zero-subtree)` still evaluates to zero, so changing `intron` into `intron'` has had no effect on fitness.

After extended evolution under crossover, large trees may be formed which contain a high proportion of repeated code [27]. These repeated patterns may include introns. Thus, large bloated evolved trees may contain multiple introns.

2.4 Background: Neutrality in Genetic Programming

Neutrality is the ability of genetic operators to generate children of identical fitness to their parents [57]. Like bloat (Section 2.2) it has a large literature. (Wright and Laue [63] include a recent summary.) However, whilst bloat is often regarded negatively, as Galvan-Lopez says "The hope is that the presence of neutrality can aid evolution" [10]. Evolution is often regarded as search through a fitness landscape [64]. If the landscape is broken, uneven or jagged, evolution might be slow or even get trapped on an isolated hill top. Neutrality can be thought of as movement along a contour or across a plateau, which, it is hoped, allows evolution to pass more easily through a rugged landscape [62]. Although, for example Renzullo et al. [48] consider neutrality in the programming domain, work on neutrality has mainly considered Boolean or discreet problems, e.g. [16], and little research has considered neutrality in the continuous domain [56]. As with Section 2.1, even existing work on neutrality in Boolean problems or digital circuits deals with small programs which are far shallower than the nested floating point expressions of depth ≈ 400 and about 3000 in Section 3 onwards.

2.5 Motivation: The Impact of Information Theory in Functional Programs on Genetic Programming Fitness Landscapes

In software engineering Voas' PIE model [59] says bugs have three components: **E)** the faulty code must be *executed.* **I)** the executed bug must *infect* (i.e. change) the program's state. For example, consider a faulty shift left with "+" instead of "*" in the statement `x=2+y`. It will produce the right value (4) when `y` is 2. In which case, since `x` is 4 in both the faulty and correct code, there is no infection or disruption of the state. Whereas if `y` is 3, `x` is set to an erroneous value (5 rather than 6). Note the program's internal state is different, i.e. it has been infected. **P)** finally the change of state must propagate to the program's output. In our experiments, we ensure that the mutated code is executed, we measure how often it infects the state and then study how this disruption travels within the highly nested function calls. (Section 5.6 considers the implications for human written programs.)

Much genetic programming work, for example classification and symbolic regression, do not include side effects, i.e. it uses pure functions. Notable exceptions are evolved agents, where the agent's actions immediately change its environment and those changes impact the agent's inputs, e.g. the Santa Fe trail artificial ant [33] and robot soccer [1].

We can view genetic programming trees as functions, whose inputs are leaves of the tree. These inputs feed information into the function. The information propagates up through the tree to emerge transformed by the function at its root node. This is true of the functions formed by the subtrees in the function. That is, the whole function is composed of subfunctions, each of which transforms the information in its inputs into the information in its output.

In today's computing almost all functions are irreversible. That is, from their return value it is impossible to infer their inputs (see also Appendix A). From an information perspective, the inputs contained more information than the function's output, i.e. information loss is inevitable. To make this concrete: consider 32 bit floating point multiplication. Each 32 bit float contains at most 32 bits of information. Multiplication takes two 32 bit values (containing at most 64 bits of information) and gives a single 32 bit output. We can think of each function as an information funnel, with a wide opening mouth to take in more information than its narrow output passes on.

In terms of information theory: classic introns lose all their information in one go. For example, with `(× intron zero-subtree)` any information provided by the `intron` has no impact as it is immediately lost by the multiply by zero.

We shall study the impact inherent in using irreversible functions. To keep as general as possible, we consider the GP landscape itself by sampling directly from it, rather than just studying evolved trees. We show information loss causes the GP search space to be smooth. With, provided they are buried deep within the tree, many crossover and mutation changes having no externally visible effect, even when they are executed many times. The high heritability of introns tends to mean they occur frequently in evolved trees [32] but since they immediately lose all their information, they are less interesting and therefore, in Section 5, we do not investigate them further.

For simplicity all the experiments below use functions with two arguments, giving rise to binary trees. However, the above arguments can be extended to other GP function sets and trees.

3 METHODS AND EXPERIMENT DESIGN

We start with trees of size of 25 001 (internal + external), and then move to trees of 1 500 001 nodes. Binary trees on average have a depth of close to $\sqrt{2\pi|\text{size}|}$ Flajolet and Oldyzko [9]. Thus the mean depth of trees of size 25 001 is ≈ 400 and the mean depth of trees of size 1 500 001 is about 3000. Our earlier work with evolved trees showed on average typically the impact of mutations was lost after it traversed about 100 functions [25]. Thus, if the effect holds in general and not just in evolved genetic programming trees, we would expect to see the effect in trees of 25 001 nodes. (As indeed we do.)

In the second set of experiments we seek to model the impact of information passing through long chains of random functions, without the confounding factor of the decay of the mutation's disruption being cut short by reaching the tree's root node. Hence the second set of experiments in still deeper trees.

We use recent code from [24], which was based on Iba's 1995 work [17, 18], to create random polynomials. Firstly the tree shape is chosen uniformly at random from all the trees of the chosen size then its internal nodes are randomly labeled with either + or ×, cf. Table 1. Its leaves are also randomly labeled[5].

For the leaves, with a probability of 50% the input X is chosen. GPquick [52] reserves 250 of its 255 available opcodes for 250 constant values. The 250 constants are chosen at random without replacement from the 2001 multiples of 0.001 between -1.0 and +1.0. The other 50% of leaves, i.e. not X, are chosen uniformly at random from these 250 constants.

To create the mutant, the parent is copied. A site in the copy is uniformly randomly selected to be mutated. The subtree at that location is removed and replaced by another subtree. The inserted subtree is similarly randomly chosen uniformly from another large random expression of the same size, cf. Koza's subtree crossover [19].

[5]Although GP trees are definitely not random, e.g. [27]: Firstly we are studying the space they inhabit but also large binary GP trees evolved by crossover may be shaped somewhat similarly to random trees [20, 21].

Table 1. Impact of Mutation on Large Random Polynomials

Terminal set:	X, 250 constants -0.995 to 0.997[a]
Function set:	MUL ADD
Fitness cases:	1501 inputs from -1.5 to +1.5 (at 0.002 intervals).
Parameters:	Trees are created uniformly at random [24]. Each mutant is created by a random subtree crossover with another random tree of the same size. I.e., as Koza's subtree crossover [19] but all positions in each tree are equally likely, therefore there is no bias in favour of functions over leaves.

[a]By chance none of the special values -1.0, 0, or +1.0 are included in the 250 randomly chosen constants.

Each pair of parent and mutated trees are both tested on 1501 test values which, for simplicity, are regularly spaced from -1.5 to +1.5. A typical GP experiment uses far fewer tests and they might be randomly drawn. For example, our polynomials are based on Koza's sextic symbolic regression problem, which uses 50 test points randomly drawn from the interval -1.0 to +1.0 [19]. Although our range of values is larger than Koza's, we still require $|X| \leq 1.5$ in order to limit the amount of numeric overflow possible in very high degree polynomials[6]. The parent and mutant are both run with X =-1.500, -1.498, -1.496 up to X =+1.500. Using our incremental evaluation [25], we can efficiently note all the places in the two trees where their evaluations differ and by how much.

4 RESULTS

We present results for trees of 25 001 nodes (Section 4.1) followed by those for trees of 1 500 001 (Section 4.2). Both sets of results are analysed in Section 5.

4.1 Experiment: Dissipation of Change

We reprise our recent experiments with ten polynomials with 25 001 components [30] but with a thousand times as many examples and more test cases (1501) before moving on to more extensive modeling and still bigger polynomials.

Figures 3 and 4 show the first ten of 10 000 examples. They show in three cases disruption is halted before reaching the root node. For these three random mutations (fun 1, 6, and 8, Figure 4) all 1501 test cases are totally dissipated before their impact can reach the root node. Therefore these three mutations are not visible externally. In the remaining seven cases disruption is rapidly quenched but does not quite reach zero before encountering the limit of the polynomial.

Notice root mean squared (RMS) difference is not monotonic (Figure 4). It can both rise and fall as we move up the tree away from the mutation site. In some cases there is a tendency for the average difference in evaluation before and after the mutation to increase with distance. This may be due to evaluation for test values below -1 or greater than +1 (i.e. $|X| > 1$) tending to increase in magnitude, as X is raised to higher powers, with corresponding increase in the difference between the evaluation in the parent and in the mutant. However, any test where the parent and child evaluation become equal, their difference must be zero and remain zero. Thus, in three cases, where all test cases are the same, the RMS difference also falls to zero.

Next we move from looking at ten trees to a sample of 10 000 trees. To make plotting feasible, Figure 5 shows just a one percent sample of them. (Section 3 described how these 25 001 node trees are created using our fast C++ code [24].)

Sometimes the mutation makes no difference, either genetically of phenotypically. For example, approximately 1 in 16 (6.25%) of mutations cause a leaf to be replaced with an identical leaf. That

[6]If much larger test values are used, we get numerical overflow and the whole function (and its mutant) returns either `-inf` or `inf`. See also Figure 17.

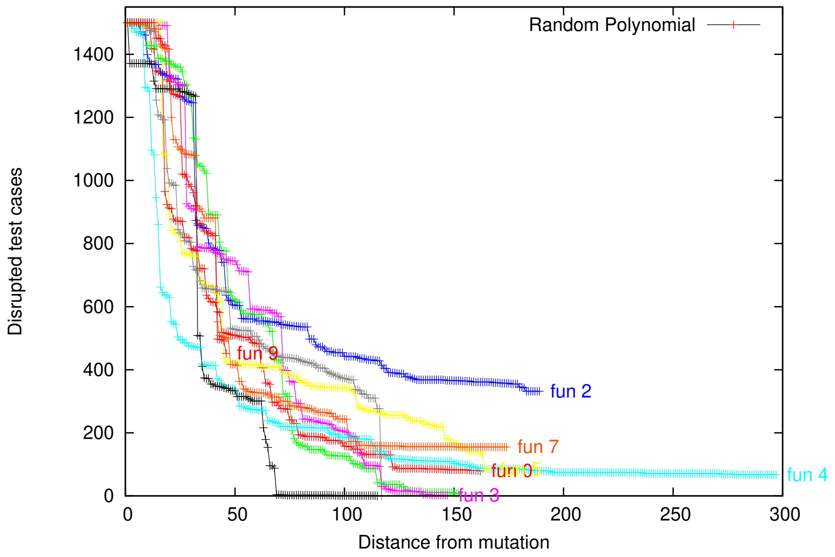

Fig. 3. Fall in impact of ten example changes with distance from disruption in random polynomial functions of size 25 001. As expected the fall in the number of test cases which detect the mutant is monotonic.

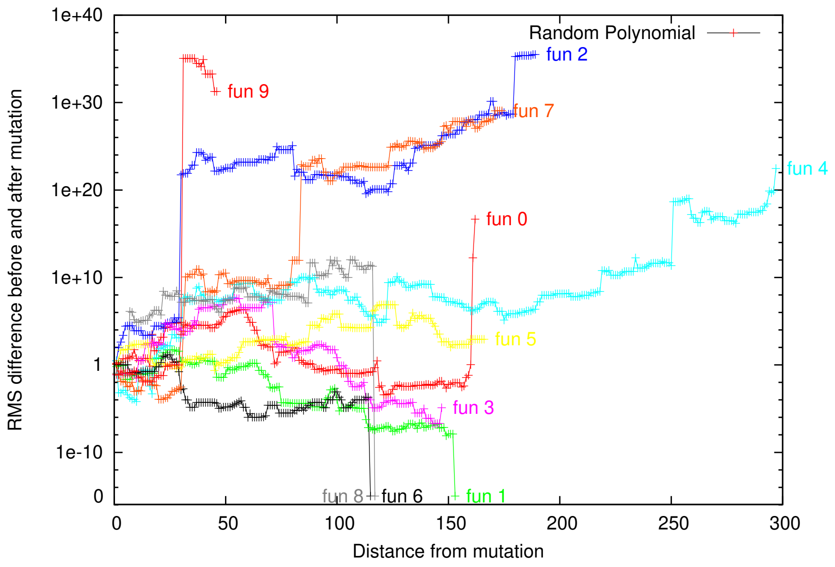

Fig. 4. Impact of the ten changes shown in Figure 3 measured by root mean squared difference on the 1501 test cases. Colours are the same as in Figure 3. Note non-linear vertical scale.

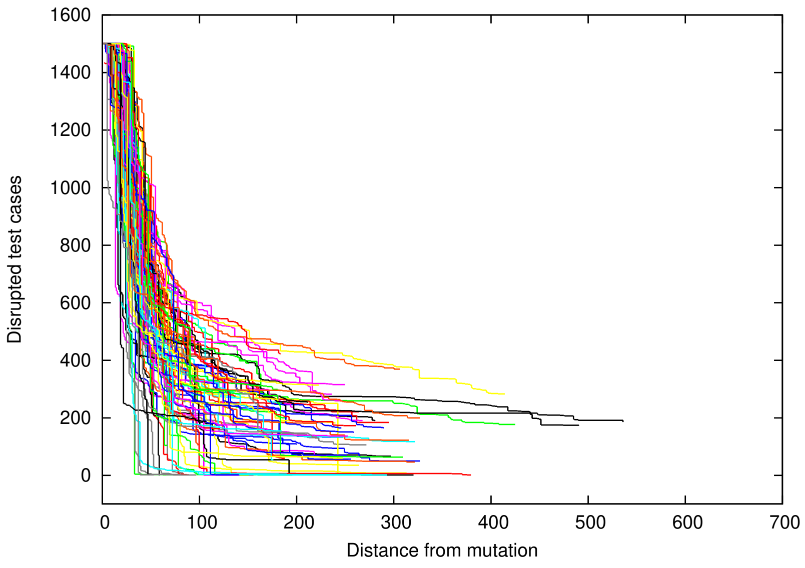

Fig. 5. 1% sample of 10 000 traces plotting number of test cases where mutant and parent evaluate differently versus distance from the mutation site. Small vertical and horizontal offsets added to separate the plots. Averages and summaries of all 10 000 are shown in Figures 6 and 7.

is, the mutation occurs but the child is identical to its parent. Rarely a mutation makes a genetic change, e.g. replacing (+ X 0.992) with (+ 0.992 X), which evaluates to exactly the same thing in parent and child. That is, there is a genetic change but no phenotypic change. In our sample, 603 mutants (6%) give no difference (and are not plotted in Figure 5).

Half of all mutations (50%) evaluate differently on all 1501 tests at the mutation site. That is, they are visible immediately on all test cases. 39% of mutations are immediately different on all but one test. The remaining 5% of mutations give identical results on between 2 and 562 tests at the mutation site.

As Figure 5 makes clear, as we analyse the impact of each mutation further from the mutation site, i.e. towards the tree's root, the number of disrupted tests falls monotonically. After 38 additions or multiplications most mutations are invisible on most tests. After 116 moves towards the root, most mutations make no difference to 90% of tests. After 124 99% and so on. However, the distribution has a long tail and only 14% of mutants are deep enough (sufficiently far enough from the tree's root node) to conceal themselves on all 1501 test cases.

In Figure 6 instead of summarising the 10 000 plots by sub-sampling, we plot the minimum, 1st quartile, median, 3rd quartile and maximum. Figure 7 plots the sample again but using log-log axis and concentrating upon the average survival of disruption. In both Figure 6 and 7 the median is plotted as the solid (red) line. Figure 7 shows over most of its non-zero range, the median is close to Zipf's law [65] (gradient -1).

4.2 Deeper Trees, 1 500 001 nodes

We extend the analysis of the previous section on random trees that are sixty times bigger (hence almost eight times deeper).

Figure 8 shows a 1% sample of our ten thousand 1 500 001 nodes trees. The immediate effects of mutants is unchanged by using the deeper trees. For example, the fraction of mutations which

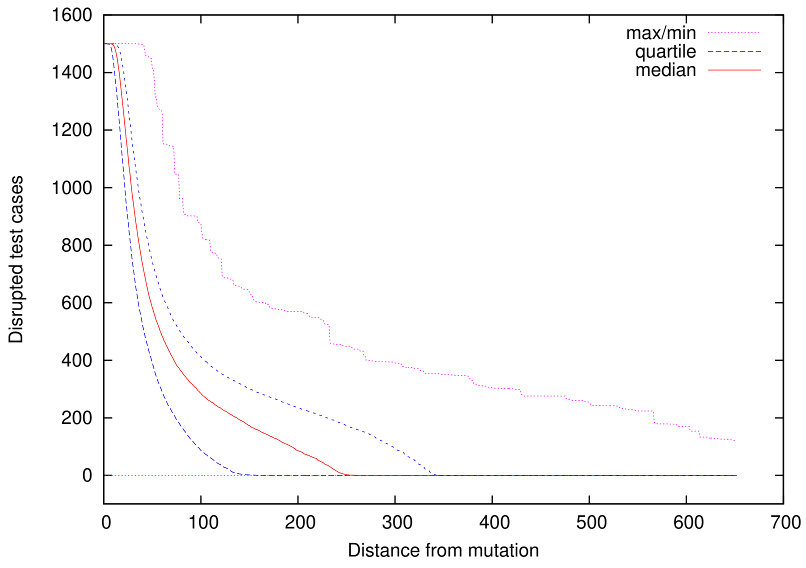

Fig. 6. Summary of 10 000 random mutations to random polynomial functions of size 25 001 showing the fall in the number of test cases (y-axis, max 1501) which are different between parent and mutant as we move up each GP tree away from the mutation site (horizontal axis). Note once all the test cases are identical, they must remain identical and so the fitness evaluation can stop.

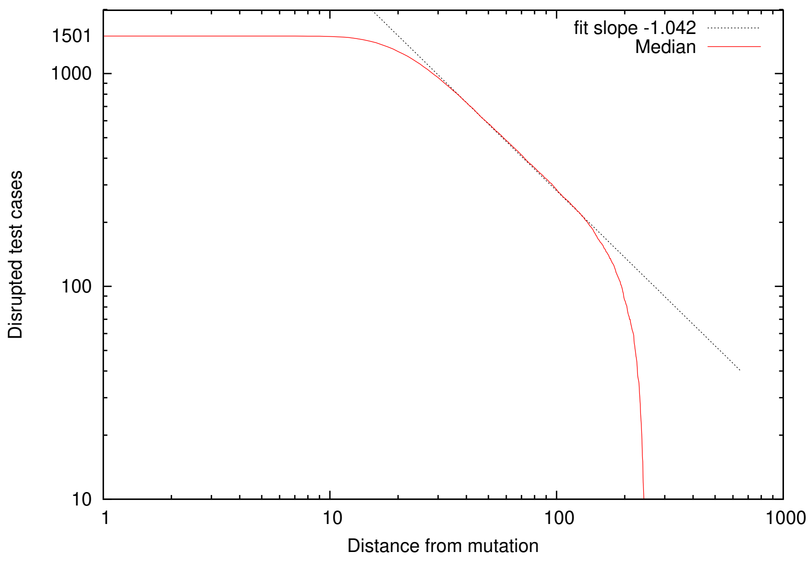

Fig. 7. Fall in average disruption with distance from mutation in 10 000 large polynomials. Dotted black line shows RMS power law fit to solid red line between x=26 and x=150 (RMS error 0.02). Slope -1.04185 is very close to Zipf's law (slope -1). Data as Figure 6 but plotted on log scales.

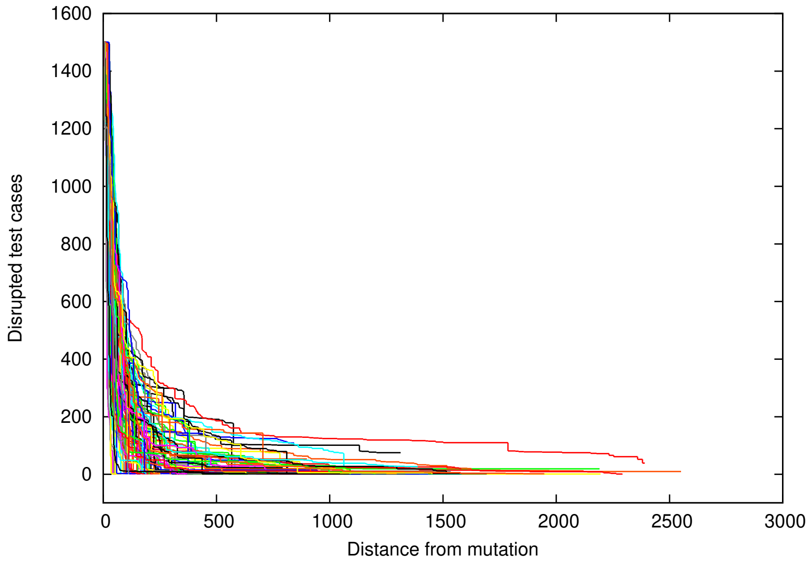

Fig. 8. One percent sample of 10 000 mutants. Each trace plots the number of test cases disrupted by the mutant against the number of nested function calls the disruption has passed through. Trees of size 1.5 10^6, cf. Figure 5. The number of disrupted tests falls monotonically. In 27.5% of cases the mutant is visible on at least one of the 1501 test cases. Small vertical and horizontal offsets added to separate the plots.

change the evaluation within the tree on all 1501 test cases after one addition or multiplication is the same as in trees of 25 001 nodes (see previous section). Similarly as we move away from the mutation site the median number of test cases where the mutation's effect can be seen falls as for trees of 25 001 nodes (again see previous section). However, at the extremes the larger trees are different. For example 274 (rather than 124) nested functions are needed for most mutants to be identical to their parents on 99% (i.e. all but 15) of the test cases. Nevertheless the larger trees did succeed in concealing most of the mutants (72.5%) on all of the 1501 test cases.

Figures 8, 9, and 10 follow the same analysis as Figures 5, 6, and 7. Whilst Figure 11 shows that, if we exclude those mutations where some of their impact did indeed reach the root node, the average number of disrupted test cases falls exponentially with distance from the mutation site, i.e. levels up the tree.

Figures 12, 13, 14 and 15 similarly follow the same analysis as Figures 5 to 11 but only include mutations which are at least 1000 levels deep. From our initial sample of 10 000 random trees of size 1 500 001, this gives us 7134 deep mutations. Figures 16 and 17 trace in detail for each of the 1501 test cases the propagation of the disruption caused by all 10 000 mutations and so explain failed disruption propagation.

Figure 12 confirms that immediately these deep mutations behave as before (cf. end of Section 4.1 and start of Section 4.2). In trees of 1 500 001 nodes, even if we exclude the shallow mutants, 16.4% of mutants deeper than 1000 are still visible on at least one of the 1501 test cases. Ignoring depth, in the full 10 000 sample 27.5% of mutants are visible on at least one test case. This suggests that perhaps there is something the exponential model is not capturing (see Section 5).

In Figure 16 we show how each mutation in the 1 500 001 node trees is impacted by each test case x. Figure 16 is almost symmetric about test value zero. It shows that the test value 0 is particularly ineffective at detecting changes. Even just one level up the GP tree, half of the mutations give values

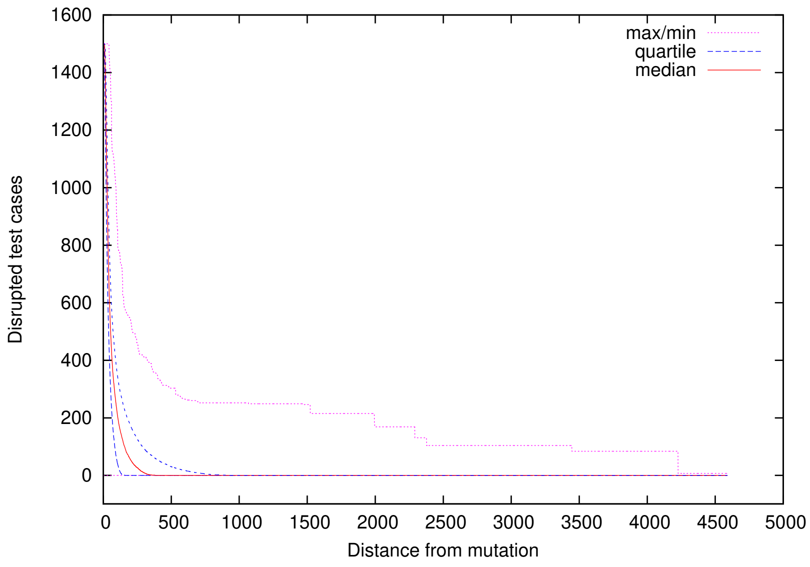

Fig. 9. Summary of 10 000 random mutations to random polynomial functions of size 1.5 10^6 showing the fall in the number of test cases (y-axis, max 1501) which are different between parent and mutant as we move up each GP tree away from the mutation site (horizontal axis), cf. Figure 6.

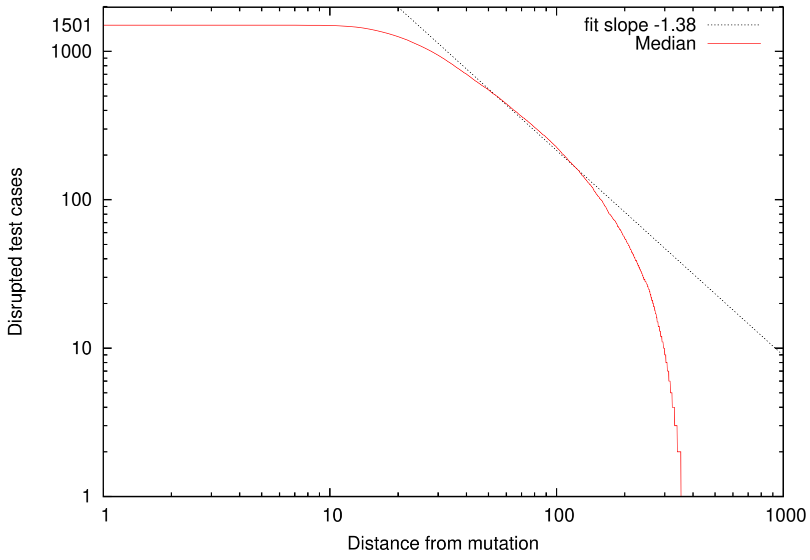

Fig. 10. Fall in average disruption with distance from mutation in 10 000 in trees of size 1.5 10^6. The dotted black line shows RMS power law fit to solid red line between x=36 and x=150 (RMS error 0.05). Data as Figure 9 but plotted on log scales. Cf. Figure 7.

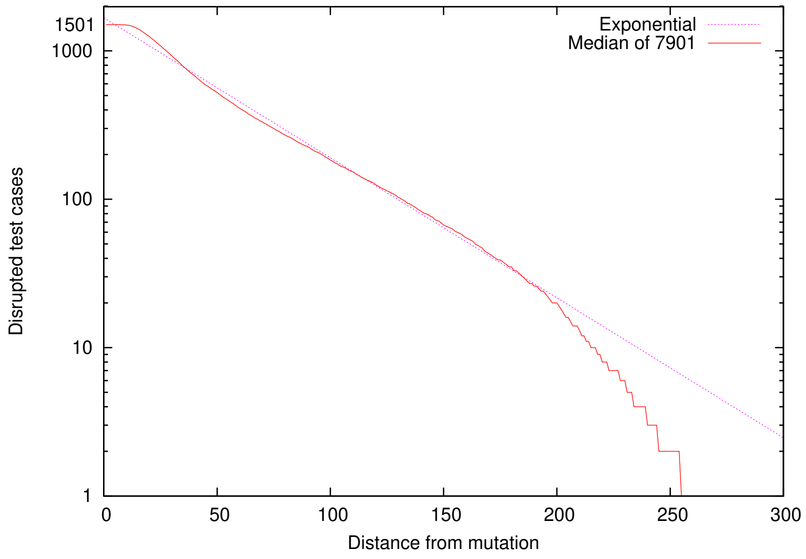

Fig. 11. Fall in average disruption with distance from mutation in 7901 trees of size 1.5 10^6 for which mutations are fully concealed. Note log vertical axis. Dotted line is best RMS fit for exponential decay between x=10 and x=200, $y = 1671.22 \times e^{-0.0217276\ x}$ (during which the average falls from 1493 to 20 tests).

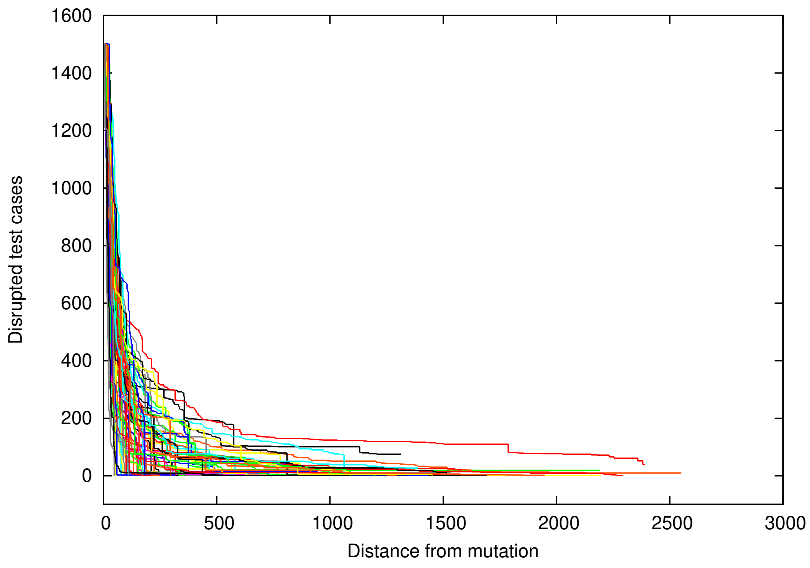

Fig. 12. Trees of size 1.5 10^6. 1% sample of 7134 plots of number of non-identical test cases between the original polynomial and its (deep) mutant. Small vertical and horizontal offsets added to separate the plots. Cf. Figures 5 and 8.

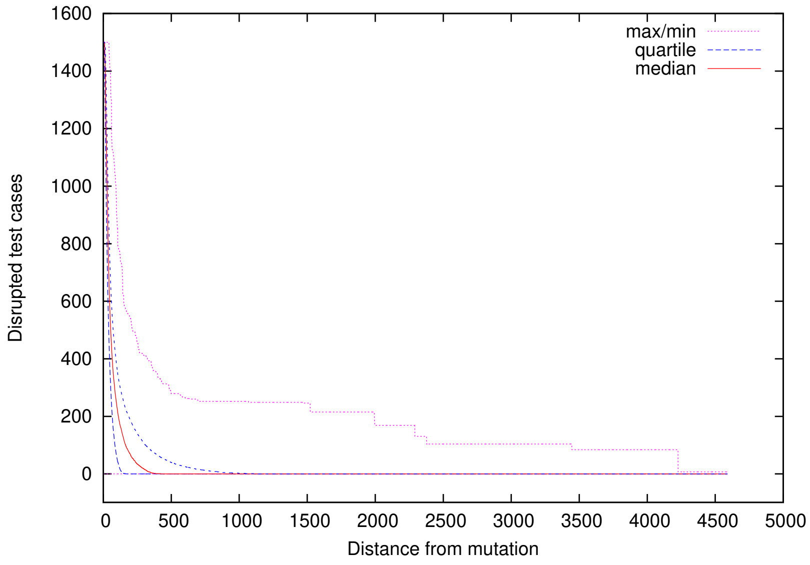

Fig. 13. Summary of 7 134 mutants at least 1000 levels deep in random polynomial functions of size 1.5 10^6, showing the fall in the number of test cases (y-axis, max 1501) which are different between parent and mutant as we move up each GP tree away from the mutation site (horizontal axis). Cf. Figures 6 and 9.

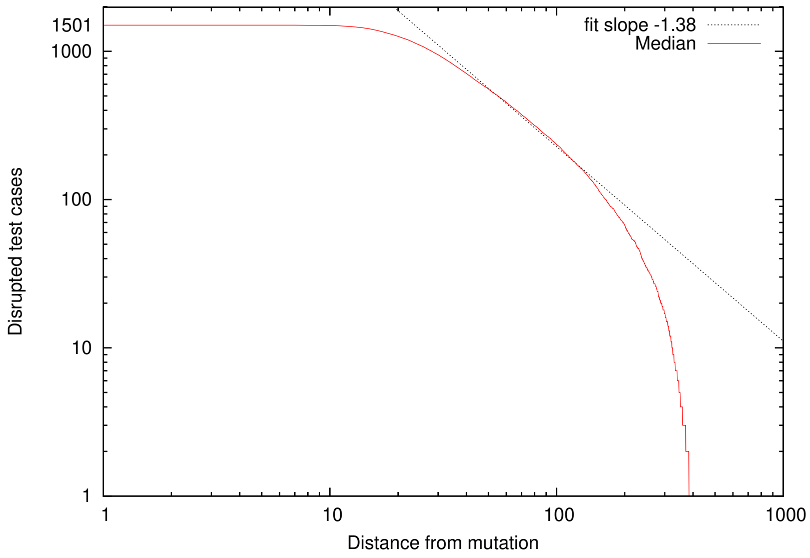

Fig. 14. Fall in average disruption with distance from mutation in 7134 trees of size 1.5 10^6, where the mutation site is 1000 or more from the root node. Dotted black line shows RMS power law fit between x=36 and x=150 (RMS error 0.04). Data as Figure 13 but plotted on log scales. Cf. Figures 7 and 10.

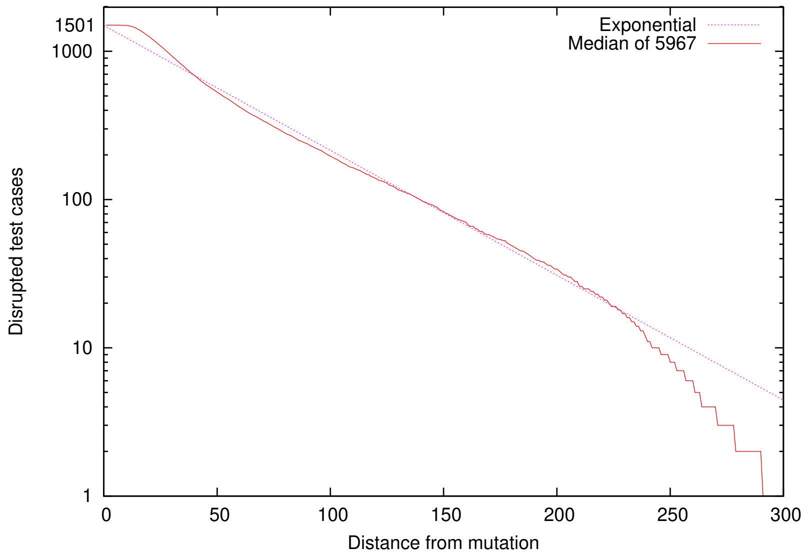

Fig. 15. Fall in average disruption with distance in 5967 trees of size 1.5 10^6 whose mutations (of depth at least 1000) are fully concealed. Dotted line is best RMS fit between x=10 and x=200 $y = 1494.15 \times e^{-0.0193952\ x}$ (during which the average falls from 1494 to 34 tests). Note log vertical axis. Cf. Figure 11.

identical to the non mutated code when tested with x=0. In contrast x=±0.002 needs nine levels, and this rises approximately exponentially with |x|. However in these random high order polynomials, on average in absolute terms the value at which evaluation of the parent and child synchronise also rises with |x|. In particular for test values outside the range -1 to +1, the synchronisation values grow rapidly and start overflowing the limits of floating point arithmetic, see Figure 17. In our pure arithmetic expressions, numeric overflow is often effective at halting the mutation's disruption and so test values $x = -1.3$ and $x = 1.2$ are the most penetrating needing on average 129 and 124 levels of nesting to be totally concealed.

5 DISCUSSION

Sections 5.1 to 5.4 present a series of models which explain various aspects of the results given in Section 4. Since the trees are uniform, the mechanisms they expose will also apply to shallow mutations but since the disruption caused by shallow mutations has less distance to travel to the program's output, shallow mutations are less susceptible to failed disruption propagation (FDP). Therefore we consider in detail deep disruptive mutations in the 1 500 001 node trees.

The exponential models below essentially assume a constant chance of an FDP event. For example, assume the chance of the disruption failing to propagate is fixed at 0.05 for each function it must pass through on its way to the root node. Then there is a 0.95 chance it passes the first function. The chance it passed two consecutive functions is 0.95×0.95, and 0.95^n for n consecutive functions. That is, the chance of propagation falls exponentially the further the disruption has to travel, i.e. with increasing n. It appears in polynomials that the mechanisms for FDP are related to floating point errors. Initially the disruption is in the region of 1.0 and it may need to progress through a few levels before, for example, floating point rounding, becomes important. We noted in Section 4.2, a "warm up" period of up to about 20 levels is needed before the exponential fit applies. We don't try and model this in detail but instead concentrate on the bulk of the curves.

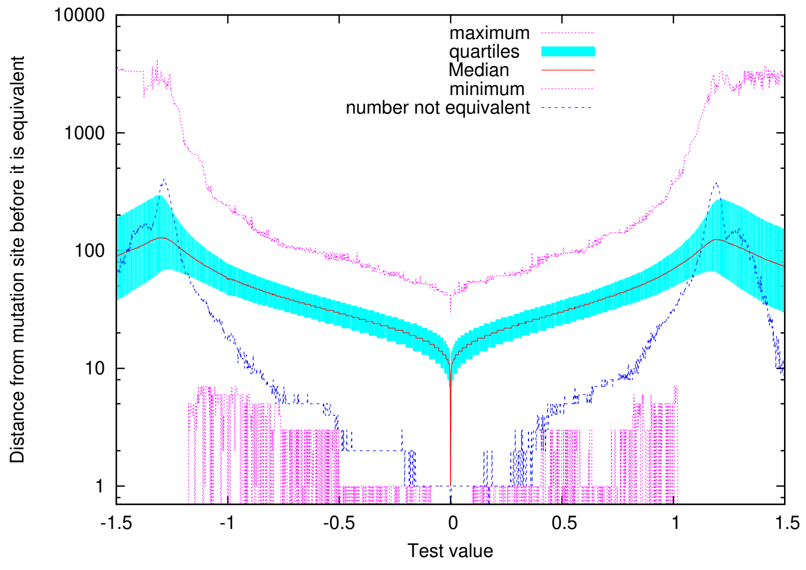

Fig. 16. Ability of test cases to expose random changes in large random polynomials (red, cyan region, purple). Trees of size 1.5 10^6. With 10 000 trials, even in large trees, except x=0 and x=0.002, all test values occasionally impact fitness evaluation (dashed blue line). The highest fraction (4%) approximately coincides with the most penetrating test values x=-1.3 and x=1.2 (red line). Note log vertical scale.

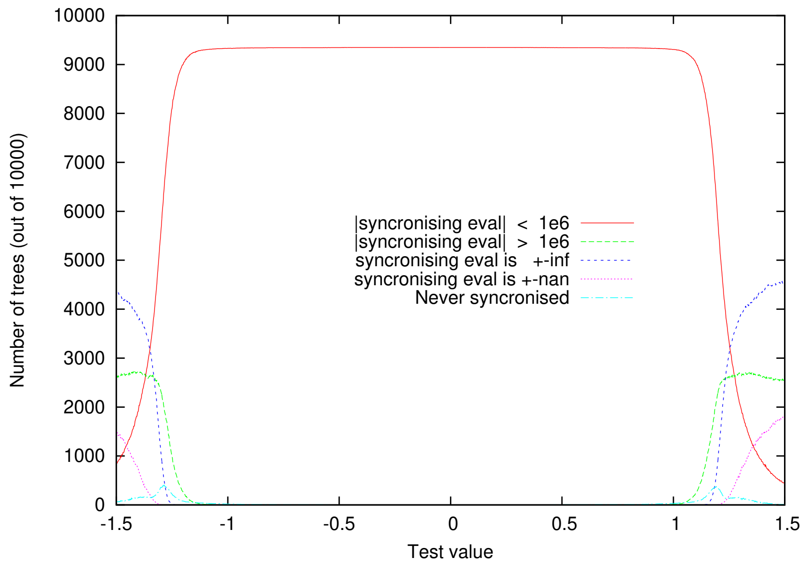

Fig. 17. Types of fitness evaluation values on which mutations' disruption stops propagating. 10 000 trees of size 1.5 10^6. For test values $|x| < 1$, almost all tests fail to reveal the change in the tree and the synchronise |value| is almost always <million (red line). With test values $|x| > 1$ mother and child evaluations tend to synchronise on increasingly large |values| (green). Until with $|x| > 1.2$, we start to see lots of floating point overflow, with evaluations synchronising on first infinity (blue dashed) and then not-a-number (purple dotted). Lowest line (cyan) shows the number of mutants where the test does indeed disrupt fitness evaluation.

Initial (unpublished) work assumed that the test cases could be treated as if they were independent and tried to fit them using a Coupon Collector [8] argument. However Section 5.2 uses Taylor series expansion to show closely packed test cases are not independent. Instead it uses a Sandpile model of $1/f$ noise, which takes into account that one event, such as the loss of disruption on one test case, may be simultaneous with similar events (the loss of disruption on adjacent test cases), and presents evidence that the effectiveness of test suites grows only slowly with n the number of tests, $O(\log(n))$.

Although information theory shows FDP must occur, Section 5.3 analyses the sequence of floating point operations just before the disruption disappears and concludes in deep floating point polynomials FDP depends upon the magnitude of the test value and occurs due to rounding and overflow errors. It finds here 0 is the least useful test value and values near ±1.3 are the best able to detect errors.

The final analysis is presented in Section 5.4, which looks at which tests are least and most effective and why and draws a connection between the Sandpile model and Zipf's law.

These models are followed by specialised discussions of their implications. Section 5.5 suggests when there is no practical limit on tree growth in extended evolution in floating point GP [28]. Whilst Section 5.6 looks at other function sets and the impact of side effects in traditional imperative programming. Section 5.7 considers other implications of information loss for software engineering and very deep artificial neural networks. These are followed by the conclusions in Section 6 and the definitions in Appendix A.

5.1 Explaining Falling Disruption

For the purposes of modeling the fading away of the impact of mutation and crossover in genetic programming with distance from the mutation/crossover site we will first concentrate upon:

- Changes that *do* make an impact at the mutation site.
  Approximately 6.25% (1 in 16) of changes involve replacing an input X with another X. There are also a very small fraction of other mutations which also make no syntactic change. Since we are primarily interested in why disruption fades, we also exclude other mutations which do change the syntax but which are phenotypically identical. For example, replacing (ADD 0.99 X) with (ADD X 0.99).
- Code changes which have no impact on any test case at the child's root node.
- Mutation sites that are at least 1000 levels deep.

As mentioned in Section 4.2, we generate 10 000 random trees of size 1 500 001. From these we reject those that do not meet the above criteria. Of the 7134 mutations which are deep enough, 487 (6.8%) make no difference to the phenotype and 1167 (16.4%) are still visible externally. This leaves 5480 mutations, which we shall analyse next (see also Figures 18 to 24).

5.2 Sandpile Model

On each test case, at each operation, it is possible that the value calculated in the mutant will become identical to that calculated in the parent. Once this happens on a particular test case (since there are no side effects), the values calculated in the parent and in the offspring will remain identical. In particular they will be identical at the root node and the change will be invisible when using that test case.

Figure 18 shows fitting an exponential curve (blue straight line) to the 5480 plots. The slope, -0.00264458, says the chance of a test synchronising at a particular ADD or MUL is p=0.264458%. The chance for an ADD is more or less the same as that for a MUL.

When test cases are closely packed, dissipation occurs in clusters [30]. It appears we can make an analogy with $1/f$ noise (also known as pink noise). Bak et al. in *Self-Organized Criticality* (SOC)*: An Explanation of* $1/f$ *Noise* [3] argue that many physical processes that give rise to $1/f$ noise and inverse power laws are analogous to Abelian sandpiles, where dropping a single grain of sand onto an existing pile of sand may have little effect or, if the sandpile is already too steep, it may trigger a cascade or avalanche of displaced sand running down the face of the sandpile. That is, a single initiating event can have consequences of arbitrary size. Bak et al. describe these critical events as self-organised because they are inherent in the system and because "no fine tuning is necessary" [3].

Although in our case, each test case can be evaluated in parallel and so the irretrievable loss of difference between the evaluation of parent and mutant on one test case cannot cause the evaluation difference to vanish for another test case, they can still be related. Indeed Figure 19 shows these synchronisation of evaluation events are not independent. If they were independent, the distribution of the number of simultaneous falls in disrupted test cases shown in Figure 19 would be Poisson distribution [60], rather than the heavy tailed power law we actually see. We suggest in order to make errors and mutations more visible, and therefore make software test suites more effective, it might be better to have independent test cases rather than simply more tests.

If we exclude the original removed subtree and the inserted mutated code, we are left with the code common to both parent and offspring. This is itself also a random polynomial. We can view it as both a function of the input X and of the values generated by the replaced R and the inserted I subtrees. R and I are themselves also functions of X. We are assuming, for a given value of X, R and I are different. However, at some point in the common code assume, for at least one test value, that the evaluation of the original and mutated code are identical. Call the polynomial at this point f(X,Z). So f(X,R)=f(X,I). As f is a polynomial it can be differentiated with respect to X. Giving another random polynomial, f′, with order one less than f, and larger co-coefficients. Consider the next test point X+δ. In our test cases δ=0.002, see Table 1. Using the first term in a Taylor expansion, we have

$$\mathrm{f}(\mathrm{X}+\delta,\mathrm{Z}) \quad \approx \quad \mathrm{f}(\mathrm{X},\mathrm{Z})+\delta\times \mathrm{f}'(\mathrm{X},\mathrm{Z})$$

Where $|\mathrm{X}| < 1$, f will be dominated by its low order components, i.e. $\mathrm{f} \approx a + bx$ and $\mathrm{f}' \approx b$.

$$\begin{aligned}
\mathrm{f}(\mathrm{X}+\delta,\mathrm{Z}) &\approx \mathrm{f}(\mathrm{X},\mathrm{Z})+\delta\times b\\
\mathrm{f}(\mathrm{X}+\delta,\mathrm{R}) &\approx \mathrm{f}(\mathrm{X},\mathrm{R})+\delta\times b\\
\mathrm{f}(\mathrm{X}+\delta,\mathrm{R}) &\approx \mathrm{f}(\mathrm{X},\mathrm{I})+\delta\times b\\
\mathrm{f}(\mathrm{X}+\delta,\mathrm{R}) &\approx \mathrm{f}(\mathrm{X}+\delta,\mathrm{I})
\end{aligned}$$

Hence (for |X|<1) it is more likely that $\mathrm{f}(\mathrm{X}+\delta,\mathrm{R}) = \mathrm{f}(\mathrm{X}+\delta,\mathrm{I})$

For $\mathrm{X} = 0$, f and f′ will both be constants.

$$\begin{aligned}
\mathrm{f}(\delta,\mathrm{Z}) &\approx \mathrm{f}(0,\mathrm{Z})+\delta\times \mathrm{f}'(0)\\
\mathrm{f}(\delta,\mathrm{R}) &\approx \mathrm{f}(0,\mathrm{R})+\delta\times \mathrm{f}'(0)\\
\mathrm{f}(\delta,\mathrm{R}) &\approx \mathrm{f}(0,\mathrm{I})+\delta\times \mathrm{f}'(0)\\
\mathrm{f}(\delta,\mathrm{R}) &\approx \mathrm{f}(\delta,\mathrm{I})
\end{aligned}$$

and so again $\mathrm{f}(\mathrm{X}+\delta,\mathrm{R}) = \mathrm{f}(\mathrm{X}+\delta,\mathrm{I})$ is more likely.

Where $|X| > 1$, f will be dominated by its high order components, i.e, $f \approx ax^n$ and $f' \approx anx^{n-1}$.

$$\begin{aligned}
f(X+\delta, Z) &\approx f(X, Z) + \delta \times anX^{n-1} \\
f(X+\delta, R) &\approx f(X, R) + \delta \times anX^{n-1} \\
f(X+\delta, R) &\approx f(X, I) + \delta \times anX^{n-1} \\
f(X+\delta, R) &\approx f(X+\delta, I)
\end{aligned}$$

So again $f(X+\delta, R) = f(X+\delta, I)$ is more likely.

We have neglected the higher order terms in the Taylor expansion of f and that R and I also usually vary as the test value X changes. So the usefulness of the Taylor approximation $f(X+\delta, Z) \approx f(X, Z) + \delta \times f'(X, Z)$ will depend on the many terms we have ignored. Detailed consideration of the second and higher derivatives together with careful consideration of rounding error could provide an upper bound on $|\delta|$ in which the adjacent test case must behave the same. Nonetheless considering the Taylor expansion gives an explanation for why the chance of nearby test points synchronising at the same point in the evaluation of the trees is higher than if they were independent.

Indeed if we want to build a test suite where adjacent tests appear to be independent when f(X,R)=f(X,I) we need f(X+δ,R) > f(X+δ,I)+ϵ or f(X+δ,R) > f(X+δ,I)-ϵ. (Where ϵ is the floating point precision.) Again detailed consideration of the second and higher derivatives could provide a lower bound on the separation of test points (δ) when adjacent test cases must have different values. For programs with continuous inputs (including integer inputs), Taylor expansion gives a potential way of reasoning how far apart test points should be and indeed how many test points we need[7]. There might be a useful analogy with shattering and Vapnik-Chervonenkis (VC) dimension in machine learning [54]. VC dimension has been used in genetic programming for controlling model complexity and regularization [11, 12] but not for designing test suites.

Figure 18 shows many examples of multiple simultaneous falls in the number of disrupted test cases. For example, even the slowest (right most) example in Figure 18 contains a case (at distance 1865) where four test cases are dissipated simultaneously. Figure 19 confirms we cannot reasonably treat tests as independent.

The exponential slope shown by the straight line in Figure 18 adds support for the view that, on average, in deep trees, where the mutation is eventually fully concealed, for our tightly packed test points; the maximum distance the mutation disruption travels in random polynomials grows as log(number of tests). That is, the impact of mutations is relatively insensitive to the number of test cases (n). Indeed dependence on O(log n) is in keeping with logarithmic dependence we previously informally argued for [30] and show in [26].

5.3 Floating Point Precision Limits Disruption Propagation in Large Polynomials

We now turn to why test cases synchronise. In the process we develop models which are specific to our experiments with large polynomials.

5.3.1 Side Subtrees. At each level, as we evaluate the disruption caused by either crossover or mutation up the tree, we have to evaluate not only the modified code but ADD or MUL's other argument. By a side subtree we mean this other argument. That is, here a side subtree is a subtree directly connected to the path from the disruption start towards the root node, but which is not

[7]For Taylor expansion to be possible, the function must not only be continuous but also differentiable. In the case of polynomials, this holds and further the number of terms in the Taylor expansion is limited by the polynomial's degree. Taylor expansion is not possible for discontinuous functions, e.g. if the evolved tree contains Boolean logic or conditional branches. In such cases establishing the degree of dependence between test points would require more sophisticated arguments, perhaps based on Walsh analysis [61] or combinatorial interaction testing [45].

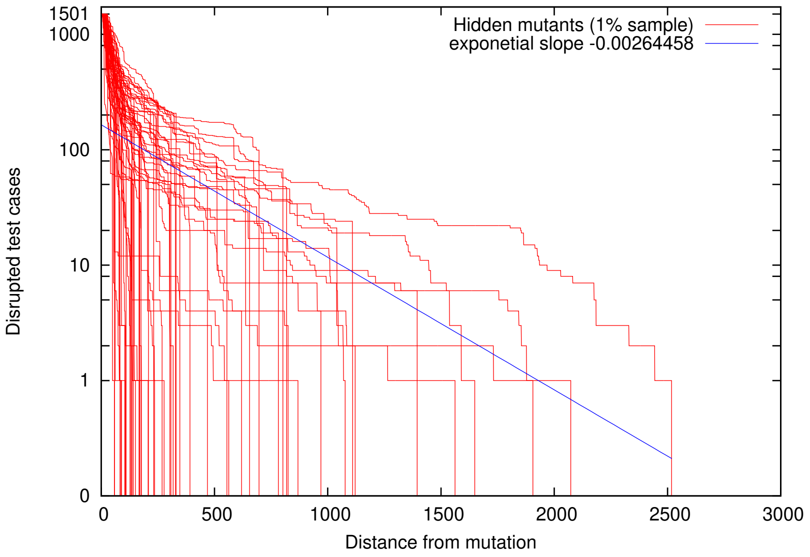

Fig. 18. Fall in disruption with distance from mutation in trees of size 1.5 10^6 for mutations (of depth at least 1000) which are fully concealed. Note log vertical axis. Solid red traces show a small sample of the data. Solid blue straight line is best RMS fit for all the data (not just the sample plotted).

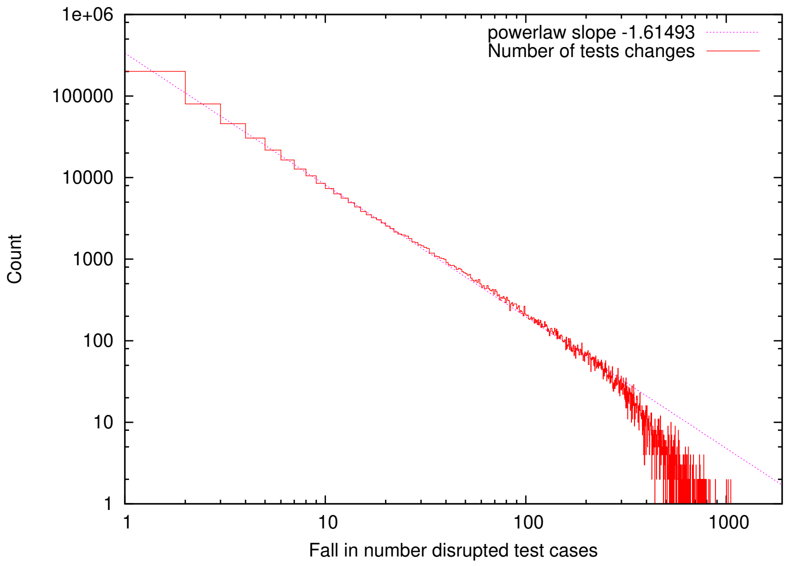

Fig. 19. Histogram of number of simultaneous falls in disruption in trees of size 1.5 10^6 for mutations (of depth at least 1000) which are fully concealed. Note log-log axis. Dotted straight line is best RMS fit, up to 300. The mean (696.3) is considerably higher than expected if the tests were independent [60]. Instead the data suggest a power law typical of $1/f$ noise and explained by the Sandpile Model [3], Section 5.2.

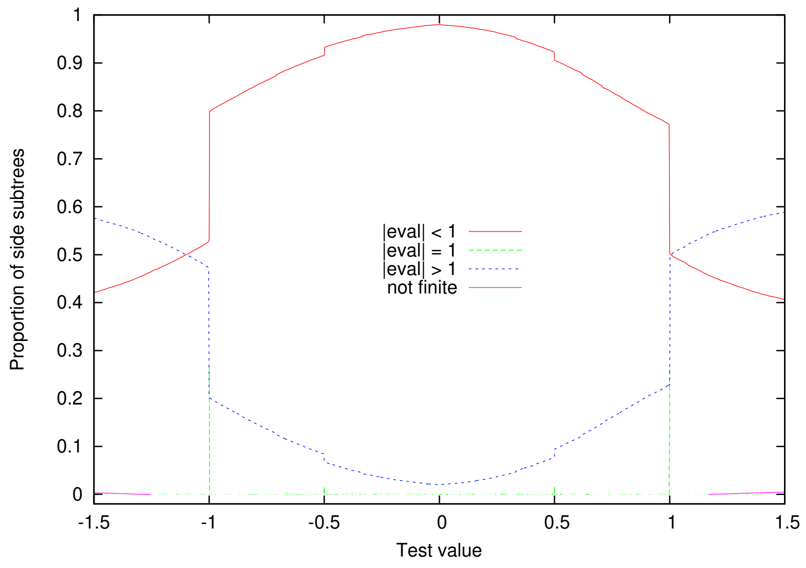

Fig. 20. Distribution of absolute values calculated by random subtrees within random polynomials of size 1.5 10^6. ±inf and ±nan values plotted together as "not finite" (lowest solid purple). Steps at -1, $\frac{-1}{2}$, $\frac{1}{2}$ and 1 correspond to where a random side subtree is most likely to evaluate to exactly -1 or 1 (dashed green line). (See also Figure 22.)

itself impacted by the disruption. As the main code itself, we have sampled randomly from the space of subtrees. For large trees, the distribution of subtrees consists of about 50% being a single random leaf and the rest being a random subtree. The random subtree distribution has a long tail of low probability deep subtrees [9]. The depth of the side trees is independent of the evaluation of the modified code. In particular, every so often, at random, the side tree will be very deep, typically corresponding to a random high order polynomial. When the input is greater than one, i.e. |x|>1, a sufficiently deep side tree's evaluation will overflow floating point precision and give either ±inf or ±nan. When such a value is either added to or multiplied by any value it will give a non-finite result (see "inf/nan" lines in Figure 21). This is true for both the original and the disrupted code. Thus encountering a deep side tree is liable to cause evaluation of both original and mutated code to become (and so remain) identical for test cases below -1 or greater than +1.

For a particular test case |x|>1, in large random trees, the chance of encountering a side subtree which evaluates to a non-finite floating point number, is random. Hence the distribution of distance from the mutation or crossover point before evaluation of the parent and the offspring synchronises will be bounded by a geometric distribution (giving exponential decay).

Note as described in the previous section, different test cases are not independent. For example, if a side tree evaluates to inf on x=1.3 it may well also evaluate to a non-finite value on x=1.4. However if a side subtree is sufficiently deep to give inf on x=1.4, its floating point operations may not overflow with x=1.3. Thus the larger |x| the more chance of encountering at random a side subtree deep enough to cause floating point overflow and so hide the presence of a mutation. (See also "inf/nan" whiskers on Figure 21.)

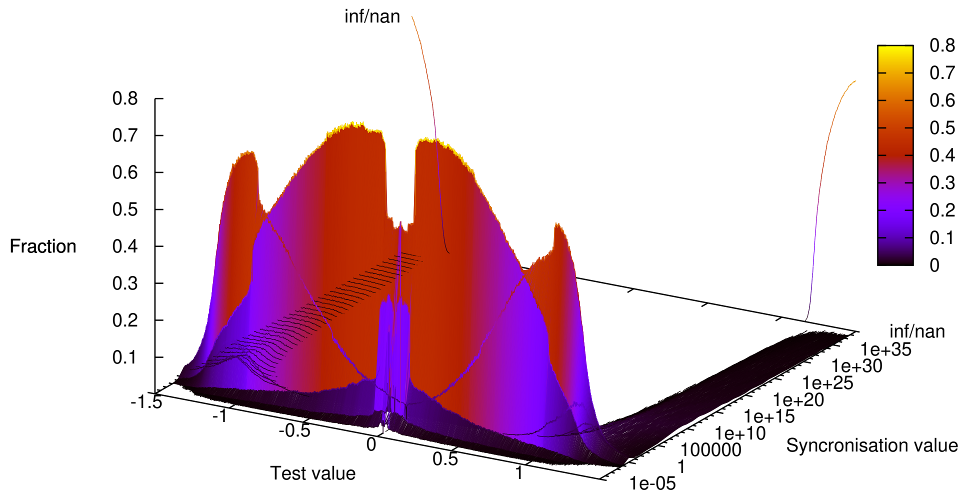

Fig. 21. Stacked histograms (lower/left – upper/right) per test case of synchronisation values at which parent and mutant evaluations are identical in large random polynomials. Data grouped into decade bins. Non-finite values placed in one "inf/nan" bin. Most test |values| < 1 are in the [0.1 to 1) bin and most test |values| > 1 are in the [1 to 10) bin. To avoid clutter syncronisation value zero and mutants where some disruption propagated are not plotted. Trees of size 1.5 10^6.

5.3.2 The Difference Between Child and Parent Evaluation When $|x|<1$. Returning to the symbols we used in Section 5.2. For a given test case $|x|<1$, the replaced R subtree and the inserted I subtree calculate values. Let D_0 be the initial difference between them, i.e.

$$D_0 \quad = \quad I - R$$

Suppose the first operation above the change point is ADD and the side subtree evaluates to S. The difference one level above the crossover or mutation point D_1 is

$$\begin{aligned} D_1 &= (I + S) - (R + S) \\ D_1 &= D_0 \text{ (ignoring any rounding errors)} \end{aligned}$$

That is, addition is a linear operation and so it makes no difference to the difference D between the evaluation of the parent and the child. Suppose instead the first operation is a multiplication. For MUL the difference one level up D_1 is

$$\begin{aligned} D_1 &= (I \times S) - (R \times S) \\ D_1 &= (I - R) \times S \\ D_1 &= D_0 \times S \text{ (ignoring any rounding errors)} \\ |D_1| &= |D_0| \times |S| \end{aligned}$$

That is, even assuming perfect implementation of multiplication and addition, the difference between the evaluation of the parent and the child will grow or shink at each MUL depending upon whether the side subtree |S| evaluates to a value greater or less than 1. For test cases where $|x|<1$, there are more side subtree evaluations which are smaller than 1 (compare solid and dotted lines in Figure 20). Therefore MUL operations tend to reduce disruption. Whilst the ADD operations have initally little effect.

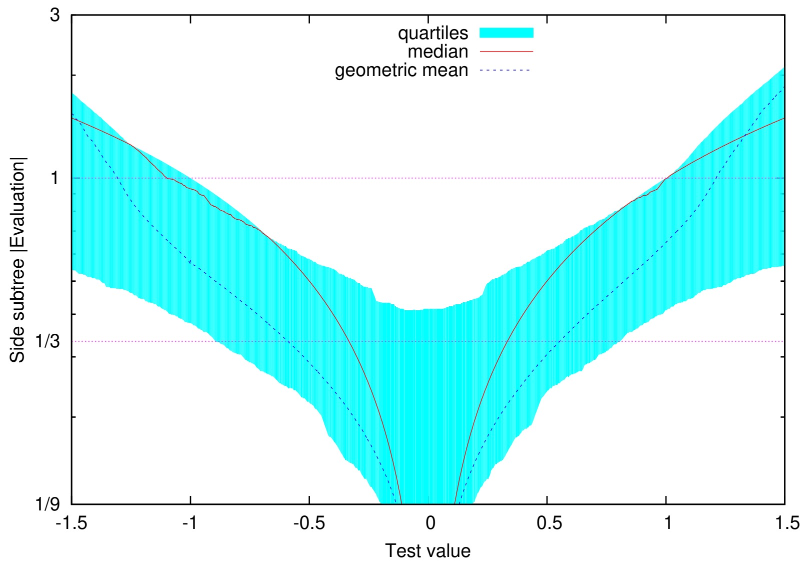

Fig. 22. Distribution of absolute magnitude of values evaluated by random subtrees in random polynomials of size 1.5 10^6. (See also Figure 20.) ±inf and ±nan values not plotted. Geometric mean (blue dashed line) calculated by excluding non-finite values and zero. Note log vertical scale.

Thus as we progress up the tree the difference D_n will tend to get smaller. If the mutation site is deep enough (n large enough), eventually we will reach an addition where the difference between the parent and child evaluation D_n will be very small compared to the evaluation of the side subtree |S| (typically |S| is about 1) and rounding error will cause the result of the ADD in the parent and the ADD in the child to give identical values.

5.3.3 Evaluating Changes When $|x| \approx 1$. We can re-use the argument from the previous section to explain why there is also synchronisation between parent and child evaluation for test cases near ±1. In fact, it is due to the asymmetry of the distribution of values calculated by the side subtree, see Figure 22. Although about half the side subtrees evaluate to less than 1 and about half to greater than 1 (cf. median, solid line, in Figure 22), near x=-1 and x=1 the upper quartile line is close to the median. This means about a quarter of MUL operations increase the difference between the parent and child evaluation but about a half reduce it (Section 5.3.2). Hence although the difference will sometimes increase it is more likely to be reduced (cf. geometric mean, dashed line, in Figure 22). If the crossover point or mutation is deep enough, floating point rounding will mean the evaluation of the parent and child synchronises and disruption does not propagate.

5.4 The Last Test Case to Synchronise

Note: although the above explanation (Section 5.3) deals explicitly with our example of floating point numbers, pretty much all computer operations are irreversible and so lose information and will cause code changes deep within a nested hierarchy of calls to be invisible to many test probes.

Figure 23 shows the absolute difference between evaluation in the parent and child on the last test case. 83% of the cases where the last operation is an ADD, it is applied to numbers in the parent and child which differ by less than 10^{-5}, although in 5.5% of cases the difference is 2.58113 10^{34} ($2^{114.314}$).

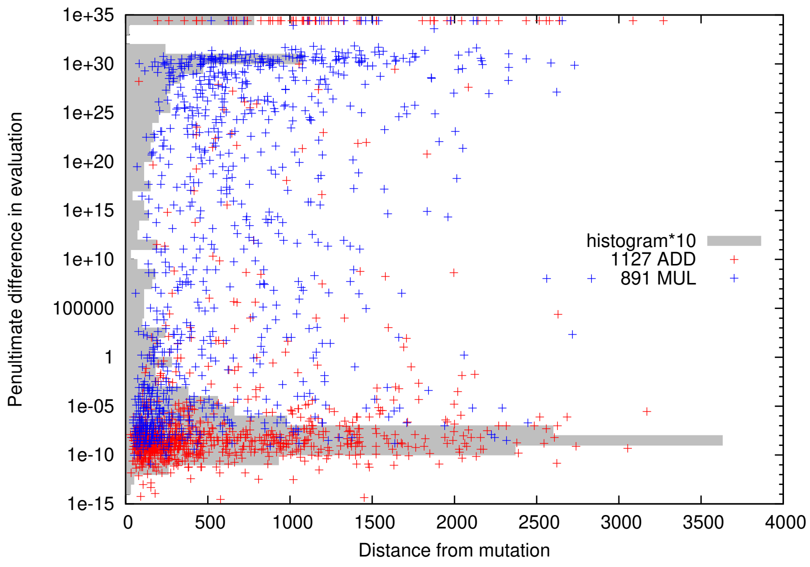

Fig. 23. Size of difference (note log vertical scale) between evaluation in parent and mutant on last test case to synchronise. Data are split by type of the operation that synchronises the remaining test case (i.e. + or ×) and by distance from mutation (horizontal axis). Trees of size 1.5 10^6 for mutations of depth at least 1000 which are fully concealed. Grey shading gives histogram on y-axis (for clarity multiplied by ten).

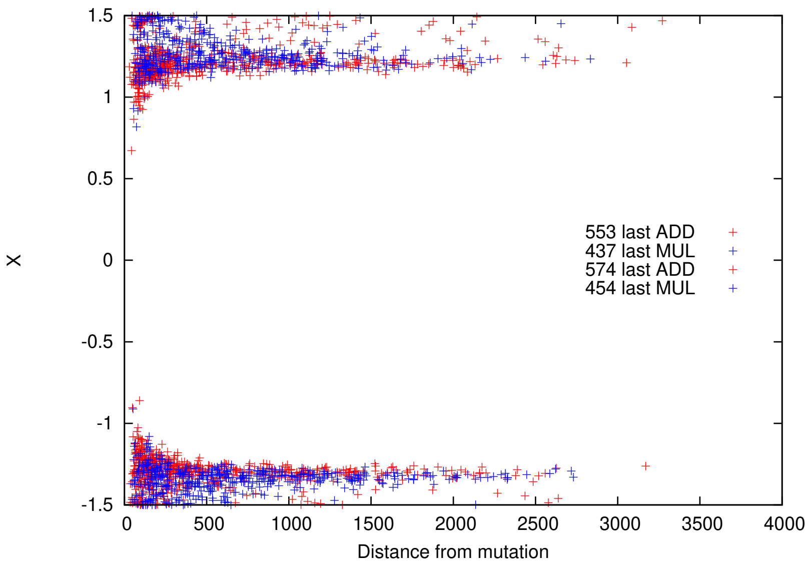

Fig. 24. Test value X in last case to be hidden. Same 2018 examples as Figure 23.

In contrast, if the final operation before all the test cases are synchronised is a MUL, the final difference between parent and child is almost uniformly spread in $\log_{10}$ space. Although again there is a small, 1.6%, cluster where the difference is 2.58113 10^{34}. In other words, most of the experimental data supports the idea that evaluation synchronisation is due to rounding error (Section 5.3.2) but, for test cases $|x|$>1.1, Figure 23 suggests there are other effects as well.

Likewise, Figure 24 concentrates upon the last test case to synchronise. It makes it very plain, as expected, that test cases where $|X| < 1$ are more dissipative (and thus synchronise faster) than $|X| > 1$. For our high order polynomials, test cases $|X| \approx 1.3$ are the most penetrative. Still larger values of |X| may lead to crossover and mutations becoming concealed and the test case failing to detect the change due to floating point overflow giving ±inf or ±nan (Section 5.3.1).

5.4.1 Test Cases Whose Disruption Propagates to the Root Node. Excluding floating point overflow and values within typical floating point tolerance (10^{-5}), in random trees of size 1 500 001 only 0.18% of tests' disruption propagates as far as the root node and thus can affect fitness, see blue + and × in Figure 25. Figure 25 plots the value of test cases which do detect code changes, the depth of the change and the size of their impact. Figure 25 confirms that most cases of significant disruption propagation occur in relatively shallow (depth≤288) code changes. It also shows (like Figure 24) that test values near ±1.3 are the most effective.

If we concentrate on effective changes more than 1000 nested functions deep we see: mostly the difference between the original and mutated output is only in the way that floating point overflow is represented (red dots in Figure 25). That is, the original tree returned ±inf or ±nan and so does the mutant, but it gives a different type of overflow. For example, the original returned -nan and the mutant returned -inf. Typically in GP such trees would each be given the same poor fitness value. The mutations deeper than 1000 which do disrupt the output by a finite value (i.e. not ±inf or ±nan) are approximately evenly split between three classes. In 31% the difference is within typical floating point precision (green dots in Figure 25). In 39% the difference is less than 1% (light blue +). In 30% the difference is more than 1% (blue ×). Of those where the difference is more than 1%, only 143 are deeper than 2000. Almost all of these involve numeric overflow (72%) and in only 2 cases is the difference more than 10% and the evaluation values have not already exceeded a million. That is, even this tiny fraction of fitness tests which do have an impact, are consistent with our earlier explanations (Sections 5.2 and 5.3) and our claim that given enough nested functions (i.e. given a deep enough tree) due to real computing effects (such as rounding error and numerical overflow) even well behaved floating point operations will lead eventually to code changes having no effect on fitness.

5.4.2 Modeling Failed Disruption Propagation in Polynomials: Zipf's Law. Figures 7, 10 and 14, show that on average the number of fitness test cases whose evaluation is disrupted by crossover or mutation falls with distance from the crossover or mutation site as a power law with a slope of $\lessapprox -1$, i.e., approximately follows Zipf's Law [65]. In Section 5.2 we showed that the events which cause the synchronisation of the evaluation of the original and the changed function and so conceal the code change from closely packed test cases are not independent. Section 5.2 uses Taylor's approximation to show rounding errors in similar fitness test cases, which may lead to evaluation syncronisation and thus failure of disruption to propagate, are indeed related. It then uses Bak et al. [3]'s sandpile $1/f$ model, to suggest the distribution will follow a heavy tailed power law, of which Zipf's -1 power law is the most famous example.

5.5 Failed Disruption Propagation as a Limit to GP Tree Bloat

In the discrete case, we previously reported [22, 23] that trees can evolve to be so large that in practice no change made in any member of the population makes a difference to fitness. Further if

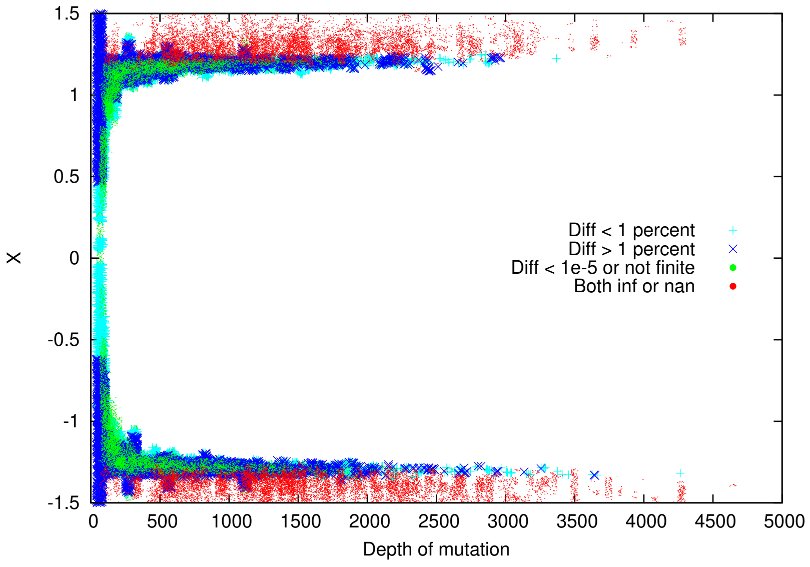

Fig. 25. 0.4% of test values X (vertical axis) which *do* impact fitness evaluation after 10 000 large random polynomials have been randomly mutated. Trees of size 1.5 10^6. Notice test values near zero only detect shallow mutations. Most disruptive mutations when tested with values |X| <1 are less than 40 levels deep (max 150). 36% of differences (red dots) are due only to numerical overflow. Green dots (21%) show tiny differences. + show (27%) other differences that are less than 1%. × show (16%) other differences which exceed 1%. Small amount of noise added to spread data.

this total fitness convergence continues for many generations bloat stops and instead the size of trees in the population executes an undirected random walk.

For the random walk end of bloat to hold, we need the chance of any change in any member of the population impacting fitness to be small. It appears, typically[8] just a single tree in the population which does not have the maximum fitness in just a few generations, is sufficient to keep trees growing on average. That is, only if there is no fitness differential driving growth [32], does bloat end to be replaced by a random walk in the tree sizes [22, 23]. Effectively the need to have on average less than one tree in the population whose fitness is damaged by genetic change, increases the shallow sensitive target area around the root node [22, 23] in proportion to the population size.

Returning to the continuous case. Figure 25 suggests for a population of the same size as our sample (i.e. 10 000), the same primitives, test suite, etc., fitness evaluation will be affected by subtree changes several thousand levels from the root node. Suggesting bloat will not be stifled until trees are much deeper than this. Figure 25 suggests the crossover depth needs to consistently exceed 4500. If the tree depth is 9000 then on average about half its internal nodes will be more than 4500 levels from its root node. A random tree of depth 9000 will contain about a 13 million nodes ($9000^2/2\pi$, Section 3). Of these about 1 in 2 are less than 4500 levels from the root[9]. For the ratio to be less than 1 in 10 000, we can make the tree 5 000 times bigger, i.e. 65 billion nodes. Having on average even just one disruptive crossover per generation appears to be enough to sustain bloat [22, 23]. To observe stagnation and no evolution this would have to be reduced to less than one in

[8]That is, in conventionally sized genetic programming (GP) populations with no anti-bloat measures and typical high fitness selection pressure, e.g. tournament selection [5],[34, page 185], an occasional single unfit tree is enough to drive evolution.
[9]On average for large random binary trees the mean depth is half the tree depth [50, Theorem 5.3 v. 5.8].

ten or even one in a hundred generations, requiring proportionate increases in the tree size, to 650 billion or even 6.5 trillion nodes.

Conversely let us try restricting our test suite to -1 to +1 and reducing our GP population to 500 trees. Figure 25 shows when the test inputs obey $|X| < 1$ there are no examples of disruptive crossover deeper than 150. Let us use the same argument as before. A random tree of depth 300 will have about 14 000 nodes ($300^2/2\pi$), about half of them will have a depth less than 150. Increasing the size by 250 fold (i.e. to 3 600 000) should mean less than about 1 in 500 random crossovers will be within 150 of the root node. Therefore we anticipate less than about one fitness disruption per generation. To reduce this to one disruption in ten or one hundred generations we might anticipate tree growth until the trees are on average 10 or 100 times bigger, i.e. about 40 to 400 million nodes.

If we repeat the argument for a population of 48, we get a tree size of 340 000 for $\lessapprox 1$ disruptive crossover per generation, suggesting bloat until about 3.4 or 34 million nodes. Given the variability of random trees[10], our simple argument is in surprisingly good agreement with experiments with a population of 48 [28, Fig. 2] (albeit using a different function set). In [28, Fig. 2] we reported bloat to trees of more than 64 million nodes.

5.6 Bound on Failed Disruption Propagation in GP and Human Written Programs

We have limited ourselves to continuous functional programming. Discrete, particularly Boolean operations, e.g. AND, OR, NAND, or "protected" functions (like protected division) commonly lose information faster than continuous operations or simple functions[11]. So we suggest, where errors propagate via nested functions, floating point polynomials represent a plausible upper bound on disruption propagation both in GP and in ordinary computer programs. In future it might be possible to extend this analysis to traditional imperative programs, with variables and side effects. At present we can only speculate that global variables might be a way of passing disruption unchanged by circumventing deep nesting. However if any data are computed upon, then in deterministic programs, that computation will lose information and so has the capacity to shield the output from disruption in the data.

5.7 Implications of Information Loss

It is tempting to speculate on how our results may cast light elsewhere. For example, the progressive loss of impact of small code changes, errors or bug fixes will mean they are difficult to detect externally. Conversely, for example when unit testing, small code changes or bugs in the same unit may be more obvious as they are close by. The progressive loss of impact with distance from the disruption could mean: 1. A bug may be hidden on many test cases but there may be other cases which do reveal it. Hence only special circumstances may reproduce the bug. 2. Many changes to the buggy code may either a) produce the same result as the buggy code, causing the belief that the change did not fix the bug or b) produce the same result as the error free code, causing the, possibly erroneous, belief that the change did fix the bug.

It seems zero is not a good test value on its own. In that, many changes produce the same value when only tested with zero. Although, in these experiments, values near ±1.3 were often the most penetrating. This is partly because, in heavily nested pure floating expressions, larger values often rapidly lead to floating point overflow. Nevertheless, with arbitrary code, values near ±1.3 might be a good starting point.

[10] For trees of about 30 million nodes the fraction that are within 150 function of the root node is almost as variable as its size (coefficient of variation = $2/3^{\text{rds}}$)

[11] For example, if both OR's inputs contain as much information as possible its output only contains 41% of the input information.

Recently deep neural networks have become very popular in machine learning. For example, although AlphaGo [51] used various networks, their total depth was up to 82 layers. The trend for increasing numbers of layers seems set to continue. Although for speed of training, the layers may have less than 32 bit floating point precision, possibly making the network more susceptible to failed disruption propagation (FDP). At present deep networks do not seem to be approaching the depths we are using here and the ANN community is well aware of the problems training deep networks of continuous functions, such as the vanishing gradient problem. However information loss and consequent FDP may be a useful perspective, suggesting as it does an ultimate limit on neural network depth.

6 CONCLUSIONS

We have applied ideas from information theory, software resilience and bug detection from software engineering to help understand the genetic programming (GP) fitness landscape [59], [29], [7], [48], [14], [35], [42], [37], [6], [13], [31]. Our approach has been to consider random mutations as if they were software bugs and to measure the effectiveness of systematic testing at discovering the bug. In terms of Voas' [59] PIE framework: our mutations are always "Executed". In almost all cases the mutation makes an immediate difference to the execution. That is, the mutation/bug "Infects" the program's execution. And we model the "Propagation" of that disruption. We use information theory to show that even something as simple as a floating point implementation of a high order polynomial loses information. In tree GP this causes the impact of the mutation to be progressively dissipated with distance from the mutation site. So that the mutation (bug) may have reduced or even no effect externally. However in traditional imperative software there are often additional routes and data paths whereby internal disruption may reach the program's outputs. Therefore, although GP and software engineering may be comprised of the same functions, with the same information loss and the same failure of disruption to propagate, we need to be cautious about extrapolating too far from a purely functional language.

We study the genetic programming search landscape by randomly sampling from it. By sampling random polynomials, we both avoid the difficulties of dealing with division (especially divide by zero) and get results on high order polynomials, which will be of wide interest [15].

We previously reported in discrete Boolean problems that eventually bloat will end due to GP population fitness convergence [22, 23]. Section 5.5 suggests simple scaling laws for the end of bloat for similar continuous GP symbolic regression. Although preliminary, these suggest a strong dependence on the test set. With many tests near ± 1.3, Section 5.5 suggests an impracticable bound[12]. That is, in practice, population wide fitness convergence will not occur and so limit bloat and instead evolution may continue indefinitely. However with tests in the range -1 to +1 and tiny GP populations, Section 5.5 suggests failed disruption propagation will be plentiful and so provides an explanation for the end of useful evolution and fitness improvement previously reported [28].

In Section 5 we showed even with very high order polynomials, densely packed test cases are not independent. Instead failed disruption propagation (FDP) [42], as measured by one test case, is liable to co-occur with other test cases. We suggested this correlation is like the self-organised criticality (SOC) sandpile model proposed by Bak et al. [3] to explain $1/f$ noise and inverse power laws common in physical systems. Conversely we suggest: 1) If you have only one test case, use a test value other than zero [30]. 2) Shallow ($\leq$ ten levels) floating point polynomials on average do not hide random errors (see Figure 7 etc.). Meaning many pure continuous floating point only programs should be easy to test. 3) Test suites used with real programs should use independent tests [40].

[12]Many GP systems cannot address trees with more than $2^{31} - 1$ nodes.

ACKNOWLEDGMENTS

This work was inspired by conversations at Dagstuhl Seminar 18052 on Genetic Improvement of Software [44]. I am grateful for the assistance of the anonymous reviewers and the Meta OOPS project.

The new GPQuick code is available in http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/gp-code/GPinc.tar.gz

REFERENCES

[1] D. Andre and A. Teller. 1999. Evolving Team Darwin United. In *RoboCup-98: Robot Soccer World Cup II (LNCS, Vol. 1604)*, M. Asada and H. Kitano (Eds.). Springer Verlag, Paris, France, 346–351. http://dx.doi.org/10.1007/3-540-48422-1_28

[2] Peter John Angeline. 1994. Genetic Programming and Emergent Intelligence. In *Advances in Genetic Programming*, Kenneth E. Kinnear, Jr. (Ed.). MIT Press, Chapter 4, 75–98. http://cognet.mit.edu/sites/default/files/books/9780262277181/pdfs/9780262277181_chap4.pdf

[3] Per Bak, Chao Tang, and Kurt Wiesenfeld. 1987. Self-organized criticality: An explanation of the 1/f noise. *Physical Review Letters* 59 (27 July 1987), 381–384. Issue 4. http://dx.doi.org/10.1103/PhysRevLett.59.381

[4] Ying Bi, Bing Xue, and Mengjie Zhang. 2020. Evolving Deep Forest with Automatic Feature Extraction for Image Classification Using Genetic Programming. In *16th International Conference on Parallel Problem Solving from Nature, Part I (LNCS, Vol. 12269)*, Thomas Baeck, Mike Preuss, Andre Deutz, Hao Wang2, Carola Doerr, Michael Emmerich, and Heike Trautmann (Eds.). Springer, Leiden, Holland, 3–18. http://dx.doi.org/10.1007/978-3-030-58112-1_1

[5] Tobias Blickle. 1996. *Theory of Evolutionary Algorithms and Application to System Synthesis.* Ph.D. Dissertation. Swiss Federal Institute of Technology, Zurich, Switzerland. http://dx.doi.org/10.3929/ethz-a-001710359

[6] David Clark, W. B. Langdon, and Justyna Petke. 2020. Software Robustness: A Survey, a Theory, and Some Prospects. Presented at Facebook Testing and Verification Symposium 2020. https://fbresearchevents.bevylabs.com/events/details/facebook-tav-symposium-division-facebook-testing-and-verification-symposium-presents-dress-rehearsal-facebook-tav-symposium-2020/

[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] William Feller. 1957. *An Introduction to Probability Theory and Its Applications* (2 ed.). Vol. 1. John Wiley and Sons, New York.

[9] Philippe Flajolet and Andrew Oldyzko. 1982. The Average Height of Binary Trees and Other Simple Trees. *J. Comput. System Sci.* 25, 2 (October 1982), 171–213. https://doi.org/10.1016/0022-0000(82)90004-6

[10] Edgar Galvan-Lopez, Stephen Dignum, and Riccardo Poli. 2008. The Effects of Constant Neutrality on Performance and Problem Hardness in GP. In *Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008 (Lecture Notes in Computer Science, Vol. 4971)*, Michael O'Neill, Leonardo Vanneschi, Steven Gustafson, Anna Isabel Esparcia Alcazar, Ivanoe De Falco, Antonio Della Cioppa, and Ernesto Tarantino (Eds.). Springer, Naples, 312–324. http://dx.doi.org/10.1007/978-3-540-78671-9_27

[11] Sylvain Gelly, Olivier Teytaud, Nicolas Bredeche, and Marc Schoenauer. 2005. A statistical learning theory approach of bloat. In *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, Hans-Georg Beyer et al. (Eds.), Vol. 2. ACM Press, Washington DC, USA, 1783–1784. http://dx.doi.org/10.1145/1068009.1068309

[12] Sylvain Gelly, Olivier Teytaud, Nicolas Bredeche, and Marc Schoenauer. 2006. Universal Consistency and Bloat in GP. *Revue d'Intelligence Artificielle* 20, 6 (2006), 805–827. http://hal.inria.fr/docs/00/11/28/40/PDF/riabloat.pdf Issue on New Methods in Machine Learning. Theory and Applications.

[13] Giovani Guizzo, Justyna Petke, Federica Sarro, and Mark Harman. 2021. Enhancing Genetic Improvement of Software with Regression Test Selection. In *Proceedings of the International Conference on Software Engineering, ICSE 2021*, Arie van Deursen, Tao Xie, and Natalia Juristo Oscar Dieste (Eds.). IEEE, Madrid, 1323–1333. http://dx.doi.org/10.1109/ICSE43902.2021.00120 Winner ACM SIGSOFT Distinguished Artifact Award.

[14] Nicolas Harrand, Simon Allier, Marcelino Rodriguez-Cancio, Martin Monperrus, and Benoit Baudry. 2019. A Journey Among Java Neutral Program Variants. *Genetic Programming and Evolvable Machines* 20, 4 (Dec. 2019), 531–580. http://dx.doi.org/10.1007/s10710-019-09355-3

[15] Michael Hopkins, Mantas Mikaitis, Dave R.Lester, and Steve Furber. 2020. Stochastic rounding and reduced-precision fixed-point arithmetic for solving neuralordinary differential equations. *Philosophical Transactions A* 378, 2166 (6 March 2020), 20190052. http://dx.doi.org/10.1098/rsta.2019.0052 Discussion meeting issue 'Numerical algorithms for high-performance computational science'.

[16] 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.

[17] Hitoshi Iba. 1995. *Random Tree Generation for Genetic Programming.* Technical Report ETL-TR-95-35. ElectroTechnical Laboratory (ETL), 1-1-4 Umezono, Tsukuba-city, Ibaraki, 305, Japan. http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/iba_1995_rtgTR.pdf

[18] 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

[19] 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

[20] W. B. Langdon. 1999. Size Fair and Homologous Tree Genetic Programming Crossovers. In *Proceedings of the Genetic and Evolutionary Computation Conference*, Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith (Eds.), Vol. 2. Morgan Kaufmann, Orlando, Florida, USA, 1092–1097. http://gpbib.cs.ucl.ac.uk/gecco1999/GP-405.pdf

[21] William B. Langdon. 2000. Size Fair and Homologous Tree Genetic Programming Crossovers. *Genetic Programming and Evolvable Machines* 1, 1/2 (April 2000), 95–119. http://dx.doi.org/10.1023/A:1010024515191

[22] W. B. Langdon. 2017. *Long-Term Evolution of Genetic Programming Populations.* Technical Report RN/17/05. University College, London, London, UK. http://www.cs.ucl.ac.uk/fileadmin/UCL-CS/research/Research_Notes/RN_17_05.pdf Also available as arXiv 1843365.

[23] William B. Langdon. 2017. Long-Term Evolution of Genetic Programming Populations. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '17).* ACM, Berlin, 235–236. http://dx.doi.org/10.1145/3067695.3075965

[24] 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

[25] 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

[26] William B. Langdon, Afnan Al-Subaihin, and David Clark. 2022. Measuring Failed Disruption Propagation in Genetic Programming. In *Proceedings of the 2022 Genetic and Evolutionary Computation Conference (GECCO '22)*, Alma Rahat et al. (Eds.). Association for Computing Machinery, Boston, USA. http://dx.doi.org/10.1145/3512290.3528738

[27] W. B. Langdon and W. Banzhaf. 2008. Repeated Patterns in Genetic Programming. *Natural Computing* 7, 4 (Dec. 2008), 589–613. http://dx.doi.org/10.1007/s11047-007-9038-8

[28] William B. Langdon and Wolfgang Banzhaf. 2022. Long-Term Evolution Experiment with Genetic Programming. *Artificial Life* 28, 2 (2022). http://dx.doi.org/10.1162/artl_a_00360 Invited submission to Artificial Life Journal special issue of the ALIFE'19 conference.

[29] 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.

[30] 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

[31] 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

[32] W. B. Langdon and R. Poli. 1997. Fitness Causes Bloat. In *Soft Computing in Engineering Design and Manufacturing*, P. K. Chawdhry, R. Roy, and R. K. Pant (Eds.). Springer-Verlag London, 13–22. http://dx.doi.org/10.1007/978-1-4471-0427-8_2

[33] 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

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

[35] Mingyi Lim, Giovani Guizzo, and Justyna Petke. 2020. Impact of Test Suite Coverage on Overfitting in Genetic Improvement of Software. In *12th International Symposium on Search Based Software Engineering SSBSE 2020 (LNCS, Vol. 12420)*, Juan Pablo Galeotti and Bonita Sharif (Eds.). Springer, Bari, Italy, 188–203. http://dx.doi.org/10.1007/978-3-030-59762-7_14

[36] Sean Luke. 2009. *Essentials of Metaheuristics* (first ed.). lulu.com. http://cs.gmu.edu/~sean/book/metaheuristics/ Available at http://cs.gmu.edu/∼sean/books/metaheuristics/.

[37] Ibrahim Mesecan, Daniel Blackwell, David Clark, Myra B. Cohen, and Justyna Petke. 2021. HyperGI: Automated Detection and Repair of Information Flow Leakage. In *The 36th IEEE/ACM International Conference on Automated Software Engineering, New Ideas and Emerging Results track, ASE NIER 2021*, Hourieh Khalajzadeh and Jean-Guy Schneider (Eds.). Melbourne. arXiv:2108.12075 https://arxiv.org/abs/2108.12075

[38] 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

[39] Alberto Moraglio. 2007. *Towards a Geometric Unification of Evolutionary Algorithms.* Ph.D. Dissertation. Department of Computer Science, University of Essex, UK. http://eden.dei.uc.pt/~moraglio/Thesis_final.pdf

[40] Justyna Petke. 2015. Constraints: The Future of Combinatorial Interaction Testing. In *2015 IEEE/ACM 8th International Workshop on Search-Based Software Testing*. Florence, 17–18. http://dx.doi.org/doi:10.1109/SBST.2015.11

[41] Justyna Petke, Brad Alexander, Earl T. Barr, Alexander E. I. Brownlee, Markus Wagner, and David R. White. 2019. A Survey of Genetic Improvement Search Spaces. In *7th edition of GI @ GECCO 2019*, Brad Alexander, Saemundur O. Haraldsson, Markus Wagner, and John R. Woodward (Eds.). ACM, Prague, Czech Republic, 1715–1721. http://dx.doi.org/10.1145/3319619.3326870

[42] 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

[43] Justyna Petke, Mark Harman, William B. Langdon, and Westley Weimer. 2018. Specialising Software for Different Downstream Applications Using Genetic Improvement and Code Transplantation. *IEEE Transactions on Software Engineering* 44, 6 (June 2018), 574–594. http://dx.doi.org/10.1109/TSE.2017.2702606

[44] Justyna Petke, Claire Le Goues, Stephanie Forrest, and William B. Langdon. 2018. Genetic Improvement of Software: Report from Dagstuhl Seminar 18052. *Dagstuhl Reports* 8, 1 (23 July 2018), 158–182. http://dx.doi.org/10.4230/DagRep.8.1.158

[45] Justyna Petke, Shin Yoo, Myra B. Cohen, and Mark Harman. 2013. Efficiency and early fault detection with lower and higher strength combinatorial interaction testing. In *ESEC/FSE'13*, Bertrand Meyer, Luciano Baresi, and Mira Mezini (Eds.). ACM, Saint Petersburg, 26–36. http://dx.doi.org/145/2491411.2491436

[46] 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).

[47] Nicholas J. Radcliff. 1993. Genetic Set Recombination. In *Foundations of Genetic Algorithms 2*, L. Darrell Whitley (Ed.). Morgan Kaufmann, Vail, Colorado, USA, 203–219. http://dx.doi.org/10.1016/B978-0-08-094832-4.50019-2

[48] Joseph Renzullo, Westley Weimer, Melanie Moses, and Stephanie Forrest. 2018. Neutrality and Epistasis in Program Space. In *GI-2018, ICSE workshops proceedings*, Justyna Petke, Kathryn Stolee, William B. Langdon, and Westley Weimer (Eds.). ACM, Gothenburg, Sweden, 1–8. http://dx.doi.org/10.1145/3194810.3194812 Best Presentation Award.

[49] Craig W. Reynolds. 1994. Evolution of Corridor Following Behavior in a Noisy World. In *Simulation of Adaptive Behaviour (SAB-94)*, David Cliff, Phil Husbands, Jean-Arcady Meyer, and Stewart W. Wilson (Eds.). MIT Press, Brighton, UK, 402–410. http://www.red3d.com/cwr/papers/1994/sab94.pdf

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

[51] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy P. Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. 2017. Mastering the game of Go without human knowledge. *Nature* 550, 7676 (2017), 354–359. http://dx.doi.org/10.1038/nature24270

[52] Andy Singleton. 1994. Genetic Programming with C++. *BYTE* (Feb. 1994), 171–176. http://www.assembla.com/wiki/show/andysgp/GPQuick_Article

[53] Robert J. Smith and Malcolm I. Heywood. 2017. Coevolving Deep Hierarchies of Programs to Solve Complex Tasks. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '17)*. ACM, Berlin, Germany, 1009–1016. http://dx.doi.org/10.1145/3071178.3071316

[54] Eduardo D. Sontag. 1998. VC Dimension of Neural Networks. In *Neural Networks and Machine Learning*. Springer, 69–95. http://www.sontaglab.org/FTPDIR/vc-expo.pdf

[55] Walter Alden Tackett. 1994. *Recombination, Selection, and the Genetic Construction of Computer Programs.* Ph.D. Dissertation. University of Southern California, Department of Electrical Engineering Systems, USA. http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/ftp.io.com/papers/WAT_PHD_DissFull_USC94_Recombination_etc_Genetic_Construction_of_Computer_Programs.pdf

[56] W. A. van Aardt, A. S. Bosman, and K. M. Malan. 2017. Characterising neutrality in neural network error landscapes. In *2017 IEEE Congress on Evolutionary Computation (CEC)*, Jose A. Lozano (Ed.). IEEE, Donostia, San Sebastian, Spain, 1374–1381. https://doi.org/10.1109/CEC.2017.7969464

[57] Leonardo Vanneschi, Yuri Pirola, and Philippe Collard. 2006. A Quantitative Study of Neutrality in GP Boolean Landscapes. In *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, Maarten Keijzer et al. (Eds.), Vol. 1. ACM Press, Seattle, Washington, USA, 895–902. http://dx.doi.org/10.1145/1143997.1144152

[58] Nadarajen Veerapen and Gabriela Ochoa. 2018. Visualising the global structure of search landscapes: genetic improvement as a case study. *Genetic Programming and Evolvable Machines* 19, 3 (Sept. 2018), 317–349. http://dx.doi.org/10.1007/s10710-018-9328-1 Special issue on genetic programming, evolutionary computation and visualization.

[59] Jeffrey M. Voas. 1992. PIE: a dynamic failure-based technique. *IEEE Transactions on Software Engineering* 18, 8 (Aug 1992), 717–727. http://dx.doi.org/10.1109/32.153381

[60] Ladislaus von Bortkiewicz. 1898. *Das Gesetz der Kleinen Zahlen.* B.G. Teubner, Leipzig. https://archive.org/download/dasgesetzderklei00bortrich/dasgesetzderklei00bortrich.pdf The Law of Small Numbers.

[61] Michael D. Vose and Alden H. Wright. 1998. The Simple Genetic Algorithm and the Walsh Transform: Part I, Theory. *Evolutionary Computation* 6, 3 (1998), 253–273. http://dx.doi.org/10.1162/evco.1998.6.3.253

[62] Thomas Weise. 2008. *Global Optimization Algorithms – Theory Application* (second ed.). Thomas Weise. http://www.it-weise.de/projects/book.pdf

[63] Alden H. Wright and Cheyenne L. Laue. 2021. Evolvability and Complexity Properties of the Digital Circuit Genotype-Phenotype Map. In *Proceedings of the 2021 Genetic and Evolutionary Computation Conference (GECCO '21)*, Francisco Chicano et al. (Eds.). Association for Computing Machinery, internet, 840–848. http://dx.doi.org/10.1145/3449639.3459393

[64] Sewall Wright. 1932. The Roles of Mutation, Inbreeding, Crossbreeding and Selection in Evolution. In *Proceedings of the Sixth Annual Congress of Genetics.* 356–366. http://www.blackwellpublishing.com/ridley/classictexts/wright.pdf

[65] George Kingsley Zipf. 1949. *Human Behavior and the Principle of Least Effort: An Introduction to Human Ecology.* Addison-Wesley Press Inc., Cambridge 42, MA, USA.

A DEFINITIONS

bloat is when during evolution the programs within the genetic programming population increase in size without commensurate (or indeed any) increase in performance.
Bloat is well studied in GP and there are several effective counter measures, such as limiting the size or depth of the tree, modifying the genetic operations of crossover and mutation to bias the generation of new offspring programs in the next generation to be smaller, modifying the fitness calculation or selection mechanism, e.g. to prefer smaller programs if two potential parent programs have the same performance and after evolution, running a clean up process to remove dead code. See also intron (below). Section 2.2

classic intron is an intron (see below) which is easily recognisable as a one of the earliest described form of intron. For example, multiply by zero, or in the logic domain, ORing with true or ANDing with false. Classic introns are obvious from looking at the GP tree. And we can easily see that part of the tree (the intron) can never have an impact on the tree's output and so has no impact on fitness. For example, if there are no side effects, (MUL 0 b) obviously evaluates to zero. Therefore b can be ignored. So b is an intron. Even if b is a large subtree, it can still be ignored. Section 2.5

Closely packed test cases. In Section 5.2 we refer to closely packed test cases to mean test cases where the test values, e.g. values of the input variable x, are close to each other. So for example we could say that test values x=0.50, 0.51, 0.52, 0.53, 0.54, 0.55, 0.56, 0.57, 0.58, 0.59 and 0.60, are tightly packed. Obviously what is considered as tight will depend upon the problem, but in this example we assume nothing dramatic happens between x=0.5 and x=0.6.

deep Although from an agricultural perspective typically trees are planted in the ground and grow towards the sky, we take the usual view in computer science and draw tree data structures with the root node at the top of the page with the leafs pointing downwards towards the bottom of the page. Hence in computer science it is common to refer to the depth of a nested data structure (such as a tree) rather than its height. Similarly, a heavily nested data structure, is said to be **deep** and parts of a tree structured program which are far from the root node are also said to be **deep**.

disrupted test A disrupted test is a test whose output has been changed. That is, the value at the root node of the program is not identical to the value it had on the same test before the program was disrupted. Section 4.1

dissipative We take the physical meaning from thermodynamics and apply it via information theory to information flow within computation. That is, we apply the name to how data is processed by the program. But note that this is abstract rather than physical and independent of the hardware the program is running on.
In thermodynamics a classic example of a dissipative process is a tornado. It is far from equilibrium and although violently mixing, the atmosphere containing it does not immediately become uniform. By **dissipative code**, we mean program code which churns up the ordered information it takes in. Thus, while the disruption caused by an error (or other cause) may be readily discerned at the error, after passing through dissipative code, the impact of the error is no longer obvious. Section 1

errors (mutations) In Section 1 we introduce the idea of injecting source code changes into existing programs, which we do in later sections. In both genetic programming and software engineering's mutation testing, these changes are called mutations. From the point of view of Monte Carlo sampling we are injecting errors into existing programs, hence **errors (mutations)**.

fitness is damaged In optimisation we often want to increase a single scalar objective measure (e.g. go faster) however quite often we want to reduce it (e.g. reduce cost). In evolutionary computation, the objective is often simply called fitness. In both cases, if we use words like up or down, we may confuse the reader unless we always make clear whether we are minimising or maximising. So instead we may use emotive words like better and worse. By **fitness is damaged** we mean that fitness has got worse, and avoid endlessly saying if we are minimising or maximising it. Section 5.5

fragile There is increasing work showing that often programs are not fragile, instead it appears software may often be subjected to small knocks, small bugs or even runtime perturbations and not fail immediately or disastrously. In contrast a fragile system is not robust, instead even small errors cause it to fail. Section 1

fully concealed mutation means that the inserted code change has no external effect outside the program on any of the test cases. That is, any run time disruption it causes fails to propagate to the root node. Section 5.2.

function is a mathematical operation which takes one or more inputs and generates an output. The mapping from inputs to outputs is fixed, in that the output for a given input is always the same. For example, the addition function given inputs 2 and 3, always yields 5. The functions available to our genetic programming system (known in GP as the function set) are given in Table 1. In programs, new functions can be made by combining existing functions. In our case the programs are trees and new functions are made by taking the output of functions and passing them as inputs to other functions higher in the tree (i.e. closer to the root node). The whole tree structure is recursive, so that larger functions are made of subtrees, which themselves may be made of other subtrees and leaves. Section 2.3

heritable As in natural biology, in artificial systems, such as genetic programming, it is common to make a distinction between the genome, meaning the set of genes directly passed from the child's parents (albeit with some mixing and/or random changes) and the child's phenotype, meaning its body and behaviour. The child's genome may in turn be passed on to its own children. The phenotype is often said to arise through the interaction of the child's genes and its environment (including help, if any, from its parents) during the course of its life.

In evolutionary computation, to try to ensure performance is **heritable**, we try to take care that the genetic operations, which are responsible for the passing, recombining and modification of genomes, interact well with aspects of the children's phenotypes which impact their fitness. Section 2.3

information hiding In Section 1 we refer to the software engineering notion of information hiding, which refers to the idea that well structured human designed code should be modular and conceal the details of its internals, so that the rest of the program can be written to use it without the programmer knowing everything about the internals of the module.

information loss occurs during a program's execution. For example, information about its inputs is progressively lost due to the use of inherently irreversible computational operations (see below). Of course the inputs may be stored and so remain available, but by information loss we implicitly refer to the active computation and in particular to the program's output. The amount of information entering a computation can be quantified in terms of Shannon's entropy. Computation (e.g. x=y+z) loses information, loses entropy, so that the information in the output x is less than in y and z together. Section 2.5

intron is part of the evolved code which, although it may be executed, it has no impact on the output of the evolved program and hence no impact on the program's performance or its fitness. In software engineering such human written code may be called dead code. See also [36]. Section 2.5

irreversible In Physics a reversible operation can as easily proceed in the forward direction as in the backward direction. Reversibility is a useful theoretical idealisation. Reversible changes releases no entropy. In reality as time moves forward everything is not reversible (i.e. is irreversible) and so does release entropy. Richard Feynman pointed out that this includes computation. However the minimum free energy required for computation is infinitesimal compared to that actually used. In practice we use Shannon's ideas of information in terms of bits rather than physical thermodynamics' entropy (which is measured in joules per kelvin). Computer programs are composed of irreversible operations which lose information. Consider the operation n=popcnt(x) which returns the number of bits in x that are set. If n is 0, we know that x was zero. if n is 32, we know that x was -1. However for any other value of n, there is some ambiguity. For example if n is 1, there are 32 possible values that x might have had. We say popcnt is irreversible because in general given its output n, we cannot say exactly what its input x was. Section 1

mutation In biology a mutation is a change in a DNA sequence. Typically a mutation is small and causes a singe base to change (a SNP) however a mutation can be bigger, i.e. affect multiple DNA bases. A mutation can be inherited and pass from a parent to its offspring, and in multi-cellular organisms, e.g. during growth, pass from one cell to its daughter cells.
Similarly in evolutionary computation, a mutation is typically a random small change (e.g. a single bit flip) to an individual's genome (often called it's chromosome) but larger and more complex mutation operators are widely used. In genetic programming, many mutation operators have been used, however typically they are localised and make a small random change to the GP tree or evolving program. GP mutation operators are usually designed to ensure that the program remains syntactically valid and to try and ensure that it still runs. Section 1

robust means that something can often withstand small, perhaps transitory, events and while its performance may degrade it does not fail catastrophically. Robust is the opposite of fragile (see above). It is widely appreciated that computer systems are robust and continue to be used and deliver economic value even though they contain many software errors (bugs) and operate with noisy communications networks and user interfaces.

simultaneous fall In Section 5.2 we talk of test cases behaving simultaneously to mean they do the same thing at the same point in the program's execution. We say simultaneous even when tests are run sequentially rather than in parallel. When their execution is done in parallel and the executions are in lock-step, the action would also occur at the same time. Thus simultaneous is a useful shorthand to mean when runs execute the same statement in the program.
In Section 5.2 we are considering the number of test cases where mutated and non-mutated code calculate different values and in particular we are interested in when this number falls, because on one or more test cases, the mutant and unchanged code now agree on the calculated value. We use **simultaneous fall** when the difference in run time execution between the original and mutant code disappears on multiple test cases at the same point in the program.

smooth A smooth fitness landscape is one in which the objective values of adjacent points in the optimisation search space are similar. Where the search space is continuous, a smooth landscape may also be differentiable. That is, a smooth fitness landscape is the opposite of a rugged one. However having a smooth landscape does not guarantee the optimisation problem is easy.
In the analogy with Topography and contour or relief maps, the objective measure, or fitness, takes the role of altitude or height and movement between neighbouring points in the search

space takes the role of horizontal movement across the undulating landscape. Parts of a search space may have different characteristics. For example, one region may be smooth and another rugged. Parts of search spaces with zero variation in fitness are called plateaus. Unfortunately the analogy breaks down when the search space has more than two dimensions (n>2) as, although still valid, the resulting n+1-dimensional landscape is difficult to visualise.

If there are multiple search operators or types of mutation, each giving rise to a different neighbourhood, then the neighbourhoods will interact, making the composite search landscape confusing. Section 1

subfunction A function may be created from multiple subfunctions. Just as a tree can be formed by joining subtrees. For example, a three input function which returns the sum of its three inputs can be composed of two addition functions, (sum3 x y z) = (ADD (ADD x y) z). The two ADD functions are just the normal ADD function, but in the example they are subfunctions of the new function sum3. Section 2.5

synchronisation value In Section 4.2 we use synchronisation to mean when multiple runs execute the same point in the program and calculate the same value. (See also simultaneous fall above.) In particular, we mean that the original and the mutated code give the same value at the same point. Once two programs become synchronised, as there are no side effects, the rest of their execution, including in particular the programs' output, must remain synchronised. By **synchronisation value** we mean the value calculated by the two runs at the point where they synchronised.

test values are the values (**X**) used by test cases. In software engineering a test case will typically consist of values for the input to the program being tested and the expected output. Since tests may be used to verify that software meets the user's requirements, they may be somewhat legalistic. For example, a test may also contain additional requirements, such as the computer used to run the test must not have more than 4GB and that the output must be completed within 5 seconds.

In Section 3 each test case consists of a **test value** for the input X (between -1.5 and 1.5, see Table 1). Since the code is random, there is no defined output, instead we are interested in whether the mutated code gives the same output as the non-mutated code.