Failed Disruption Propagation in Integer Genetic Programming*

William B. Langdon
W.Langdon@cs.ucl.ac.uk
Department of Computer Science, University College London
London, UK

ABSTRACT

We inject a random value into the evaluation of highly evolved deep integer GP trees 9 743 720 times and find 99.7% of test outputs are unchanged. Suggesting crossover and mutation's impact are dissipated and seldom propagate outside the program. Indeed only errors near the root node have impact and disruption falls exponentially with depth at between $e^{-\text{depth}/3}$ and $e^{-\text{depth}/5}$ for recursive Fibonacci GP trees, allowing five to seven levels of nesting between the runtime perturbation and an optimal test oracle for it to detect most errors. Information theory explains this locally flat fitness landscape is due to FDP. Overflow is not important and instead, integer GP, like deep symbolic regression floating point GP and software in general, is not fragile, is robust, is not chaotic and suffers little from Lorenz' butterfly.

KEYWORDS

genetic programming, information loss, information funnels, entropy, evolvability, mutational robustness, optimal test oracle placement, neutral networks, SBSE, software robustness, correctness attraction, diversity, software testing, theory of bloat, introns

ACM Reference Format:
William B. Langdon. 2022. Failed Disruption Propagation in Integer Genetic Programming. In *Genetic and Evolutionary Computation ConferenceCompanion (GECCO '22 Companion), July 9–13, 2022, Boston, MA, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3520304.3528878

1 FAILED DISRUPTION PROPAGATION

If a deep GP tree is perturbed, the disruption has to propagate from the crossover point, mutation or error, up the tree through many levels to the root node before it has any impact on the tree's fitness. It is known in conventional programming [1, 22] that often disruptions fail to propagate. We argue that this stems directly from information loss and so is inherent in all computation, including GP. We have shown that failed disruption propagation (FDP) can be common in deep floating point expressions [16]. The mechanisms which cause FDP can vary between programs. For example in GP symbolic regression, FDP is often associated with rounding errors [16]. To show an example which is independent of floating point rounding, we will show (in **Section 3**) that failed disruption propagation also occurs in deep integer GP trees, where calculations are performed exactly. We find small (+1) and large (RANDINT, Section 3.6) run time disruptions rapidly dissipate in similar ways. **Section 2** describes the integer Fibonacci Problem we will use and **Section 4** discusses our results and their implications for GP and software more widely.

1.1 Disrupting Integer Arithmetic

Much of Koza's first GP book [6] is concerned with either floating point or Boolean expressions. However Koza also introduces the problem of inducing an integer tree which recursively generates the Fibonacci sequence of positive integers. Therefore we shall use the Fibonacci Problem and demonstrate in deep GP trees failed disruption propagation can also be common in exact arithmetic.

We use GP to evolve large GP trees by simply running to 1000 generations rather than stopping at the first solution. We then re-evaluate the whole tree on all the training cases having first inserted a fixed run time perturbation at a given location. We step though every location in this large highly evolved fit tree and keep a record of which perturbations do or do not cause a change in evaluation at the root node and on which test case.

Like Danglot et al. [4], the first perturbation is simply to add 1 to the evaluation (on each test case) at the chosen point in the tree. This can be thought of as the minimum perturbation. We also repeat the experiment but instead of making a small change we simply totally replace the original evaluation by a randomly chosen 32 bit signed integer value.

2 KOZA'S FIBONACCI BENCHMARK

The Fibonacci sequence 1, 1, 2, 3, 5, 8... is an infinite sequence of positive integers, where the next one is given by summing the two previous items in the list. For training we take the first 20 members of the sequence, i.e. from the 0th (1) to the 19th (6765). See Table 1. The primitives Koza [6] uses are the four small integers 0, 1, 2 and 3, the sequence index J, the three integer arithmetic operations: addition, subtraction and multiplication. In addition, to support recursion, there is a special function SRF which also takes two arguments. SRF allows access to values calculated by the GP tree on *earlier* test cases. SRF's first argument is the number of the test case. SRF's second argument is a default value, to be used if the first argument is invalid. (For simplicity all arguments, including where we have multiplication by zero, are always evaluated.) As an example (SRF 1 0) will evaluate to 0 on the first two (J=0 and J=1) test cases, and will evaluate to the evolved individual's answer for test case J=1 on later tests. Test cases are always run in order, starting at J=0. Notice, in Section 3, where tree evaluations are deliberately disrupted, the disruption is applied per test case and therefore SRF can access the earlier unchanged tree evaluations.

[*] A longer version is available on arXiv [13].

GECCO '22 Companion, July 9–13, 2022, Boston, MA, USA

ACM ISBN 978-1-4503-9268-6/22/07.
https://doi.org/10.1145/3520304.3528878

Table 1: GP to create deep fit Fibonacci trees for Failed Disruption Propagation (FDP) experiments

Terminal set:	J, 0, 1, 2, 3
Function set:	ADD SUB MUL SRF
Fitness cases:	First 20 members of the Fibonacci sequence.
Selection:	Fitness = $\sum_{J=0}^{19} \lvert GP(J) - Fibonacci_J \rvert$. I.e. the sum of the absolute error between GP's answer and the value of the J^{th} member of the Fibonacci sequence. Tournament size 7.
Population:	Panmictic, non-elitist, generational.
GP parameters:	Initial population of 50 000 trees created by ramped half and half [6] with depth between 2 and 6. 100% unbiased subtree crossover. 1000 generations. No size or depth limit.

Table 2: Ten Deep Fit GP Fibonacci Trees

Size	Depth		Fitness	Output disruption on any test			
		[7]	sum |error|	+1		RANDINT	
86035	663	735 (160)	20	0.114 %	-0.31	0.092 %	-0.31
4347	160	165 (36)	10	1.449 %	-0.30	1.449 %	-0.33
23289	220	383 (83)	184	3.010 %	-0.27	3.053 %	-0.27
131159	449	908 (197)	130	0.127 %	-0.28	0.121 %	-0.29
77479	454	698 (152)	632	0.253 %	-0.20	0.256 %	-0.20
51697	626	570 (124)	0	0.056 %	-0.27	0.056 %	-0.27
771	33	64 (14)	0	7.523 %	-0.21	7.523 %	-0.22
35727	425	474 (103)	0	0.073 %	-0.30	0.073 %	-0.30
53305	485	579 (126)	0	0.032 %	-0.33	0.032 %	-0.33
23377	360	383 (83)	0	0.137 %	-0.26	0.137 %	-0.26

3 EXPERIMENTS

3.1 Tuning for Tournament Selection

We did a series of tuning GP runs to choose the population size (Table 1). We chose a population of 50 000, where about 47% of runs find programs which pass all 20 Fibonacci fitness tests by generation 50.

3.2 Ten Extended Runs to 1000 Generations

In order to get deep fit Fibonacci trees to try our disruption experiments on, we ran our GP with a population of 50 000 for 1000 generations. (We used the same parameters, Table 1.) Of course without size or depth limits, the GP bloats [19] (see Figures in [13]). At the end of each run a Fibonacci tree was selected for perturbation. The runs are summarised in Table 2. (Column 3 gives the expected depth for a random binary tree of a given size, column 1, whilst column 4 gives the standard deviation [5].)

To reduce run time we used our [11] incremental and "fitness first" [9] evaluation. Of course this does not effect the course of evolution but reduces the number of opcodes evaluated by between 27 and 41 fold.

3.3 Fraction of +1 Disruptive Perturbations

In almost all locations in the ten deep highly evolved GP Fibonacci trees, the +1 disruption at run time fails to reach the root node on all twenty training cases. The right hand side of Table 2 gives the fraction (%, column 6) which disrupt fitness on any of the twenty test cases. Even for the three shallowest trees more than 90% of the perturbations fail to propagate to the program's output on any test case. Therefore the fitness would be identical despite the runtime change. In most cases the fraction of FDP on any of the fitness cases is well in excess of 99% (max 99.968% for run 9).

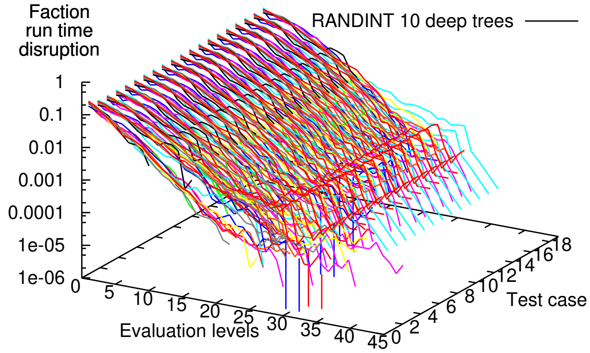

Figure 1: How far RANDINT disruption travels up tree for each training case (0–19). Small fraction which reach the root node on any test are given in Table 2 and not plotted here. Cols 7&9 of Tab 2 also give median slope. Note log scale.

In floating point symbolic regression [16] failed disruption propagation depends on the magnitude of the test case (with values near zero being suppressed most easily). In contrast, in the Fibonacci Problem we see almost the same behaviour for all test cases. This suggests Fibonacci test cases are not independent. Which in turn hints at potential runtime saving by sub-sampling test cases. Only the zeroth test case (J=0) is slightly different, and we see disruption dying away even faster than in the other nineteen test cases. Also FDP behaves similarly across ten very different trees. Each shows, for each test case, a similar exponential decrease in the number of functions the +1 disruption propagates through before becoming totally lost. Column 7 in Table 2 gives the exponent for the decrease with depth averaged (median) across all twenty test cases. The values lies between -0.33 and -0.20, meaning on average between 14 and 23 nested functions will reduce disruption by 100 fold.

3.4 Location of +1 Disruptive Perturbations

The small fraction of +1 disruptions which reach the root node, on any test case, and so do change fitness, lie close to the root node itself. I.e, the disruption travels only a short distance through the code. Depending on run, most lie within 4–9 levels of the root node. For most runs more than 99% of these disrupted evaluations start within 8 nested function calls of the root. Runs 1, 3 and 5 have long tails, so that more than 1% of disrupted evaluations start more than 20 levels deep. Even so in these runs no disruption traverses more than 31 levels and still impacts fitness.

3.5 Fibonacci Mechanisms for Failed Disruption Propagation

Although we argue that information theory suggests that failed disruption propagation is universal [13], we can see specific mechanisms for FDP in the Fibonacci Problem. For simplicity let us just consider cases where the disruption on all 20 test cases stops at the same point. 20% of such +1 disruptions stop on a multiply by zero. That is, one argument of multiply is zero and so disruption to the other fails to propagate past the multiply, since its output is zero regardless of the disruption. All the others stop on a SRF function. Of these most (98.7%) are because the SRF node returned its default value as it did before the disruption. This may be because the SRF function's first argument (the index) was already invalid and so caused SRF to return its default value (the 2nd argument). And if disrupting the index value still leaves it invalid, the SRF will continue to return its unchanged default value. Meaning the disruption stops at the SRF node.

3.6 RANDINT Disruption

We repeated the above experiments replacing the small perturbation, which simply added one to the evaluation of each point in each large GP tree for each test case, by replacing the existing evaluation by a random integer value. The value was chosen uniformly at random from all 2^{32} possible signed integer values. Then as before we trace how far the large disruption propagates through the evolved tree.

Starting with Table 2 we see that the large RANDINT disruption behaves very similarly to the +1 disruption. Comparing the last four columns of Table 2, we see almost all large disruptions fail to reach the root node and so make no difference to the program's output or fitness. And further the exponential rate (between $-1/3$ and $-1/5$) with which the average disruption dies away with distance from the root node, although different between runs, is very similar for +1 and RANDINT. Figure 1 shows the how far the large disruption travels for each test case and in each run. The log vertical scale in Figure 1 emphasises the exponential dissipation of the run time perturbations as they travel up the tree through the evolved code towards the root node. The distances traveled by the small (+1) and large disruptions are similar (see also figures in [13]). The small fraction of large and small disruptions which are able to reach the program's output are similarly clustered close to the output itself (the root node), see Figure 2, next section.

3.7 Disruptable code lies near the output

Figure 2 (To reduce clutter, SRF functions are shown with =.) uses Daida's circular lattice [3]. This can be thought of as viewing the binary tree from the top looking down on the root node (in the center) with each side subtree spread out around it. The colours indicate how many times the program's answer is changed when evaluation at that point in the program is disrupted on each test case.

As expected, the root node (center) is always disrupted and so is shown in bright yellow (20 test cases). The number of test cases disrupted falls monotonically with distance from the root node. Dotted gray lines indicates parts of the tree where disruption does not reach the root node on any test case. For ease of comparison, each plot is shown at the same scale with parts of the tree outside the square box $(-10:+10)^2$ centered on the root node not being plotted. In half the runs, this box captures all the parts of the tree where disruption does reach the root node. Indeed in all runs all the heavily disrupted parts (bright yellow) are plotted. Only in the third run, is there code which is disruptive on more than three cases (blue) which is not plotted because it lies outside -10:+10.

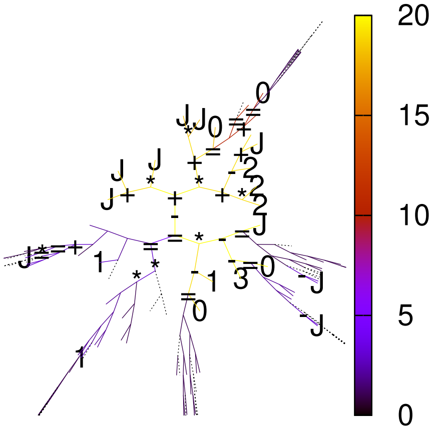

Figure 2: Colour shows locations where run time disruption impacts output of a large evolved GP tree. Brighter colours mean more test cases (1–20). Tree plotted with output at center of circular lattice [3]. Almost all the tree is grey, meaning no impact, but only the center -10:+10 by the root node is shown. = indicates SRF node, * multiplication.

4 DISCUSSION

At first sight it seems surprising in a continuous domain (albeit with exact arithmetic) that arguably the smallest (+1) and largest (RANDINT) disruptions should behave almost identically. However the two problem dependent mechanisms for failed disruption propagation (identified in Section 3.5) apply to both small and large changes. That is 1) multiplication by zero, gives a zero result, no matter how small or large the other argument is and 2) SRF will return its default value if its first argument is just out of range or if it is widely out of range.

With the small disruption and 32 bit arithmetic, integer overflow seldom occurs and if it does, it makes no difference to failed disruption propagation [13]. With the large change, overflow is more common but it still make little difference to FDP.

We have used various recent efficiency improvements to Singleton's GPquick [8–10]. In particular with extended runs [17] incremental evaluation [11] again gave substantial speed up without affecting evolution.

Columns 3 and 4 of Table 2 show again [14], but now over an extensive period (1000 generations rather than [14]'s 50 or 75), with two arity functions and no size or depth constraints (Table 1) that GP evolves trees whose shape is similar to that of most binary trees [5].

We see highly evolved integer GP programs, like floating point [15] and human written code [18] [2], are not fragile. Indeed they are robust not only to genetic changes like crossover, but to run time errors, which they did not encounter during training.

4.1 Lorenz' butterfly need not trouble software

Edward N. Lorenz (1972) was uncertain if a single flap of a butterfly's wings in one hemisphere could cause a dramatic change in the weather the other side of the equator but argued that the atmosphere is chaotic and so difficult to predict in the short term [20].

We have argued that deterministic programs are not chaotic. We have shown often a single perturbation deep within such software has no effect. Whereas the atmosphere's chaos is powered by all the Sun's energy falling on the Earth and contains an unmeasurable number of flapping wings, deterministic programs (in particular GP) dissipate information and so (mostly) give the same result even if a "butterfly" or other "bug" flips some bits deep within them [4].

4.2 Good and bad failed disruption propagation

The impact of failed disruption propagation is profound. It is a two edged sword. One side means crossovers, mutations, perturbations, radiation, coding errors, etc., may only have local impact and their disruption may monotonically fall to nothing. Making the software robust. The other edge cuts the tester: Even if an error, glitch or bug infects the local state [23], if it is far from the tester's software or hardware probe the disruption may have faded away before it can be recorded. Thus rendering the test ineffective and leaving the error undetected. However although undetected now, possibly it may have an effect on a customer later.

4.3 Better Evolutionary Computation?

In tree GP [6], almost all crossovers or mutations occur away from the root node. In deep trees, their impact is often lost before it reaches the program's output (the root node). In artificial systems we are free to choose where to place mutations and where to cut and slice in recombination. So we could opt to place such disruption close to the root node. However, if we are to evolve complex programs with many many features, they will have to be large.

Evolving a monolithic one or two dimensional structure with all information channeled via a single output node risks the program being so deep that it is impossible to measure the fitness of most genetic updates, or, if we move the crossover locations to be by the output node, we have the problem of carrying a large dead weight of code which cannot adapt beneath a tiny living evolving surface near the route node. Therefore instead we may want to adopt a porous open sponge like high dimensional structure, with a large surface area, where much of the program (and hence most of the genetic locations) is close to the program's environment [12].

5 CONCLUSIONS

We have measured software engineering's failed disruption propagation (FDP) [22] in genetic programming and find as predicted by information theory in integer functions it is very common. On average in our deep trees 99.7% of large run time disruptions fail to propagate to the root node and so have no impact on fitness.

We see failed disruption propagation scaling at between $e^{-\text{depth}/3}$ and $e^{-\text{depth}/5}$, meaning the chance of detecting disruption (be it induced by crossover, mutation, cosmic ray or indeed software bug) falls significantly within 3 to 5 levels. Indeed, with every extra level of nesting, the effectiveness of optimal test oracle placement or fitness measurement, falls by between 18% and 28%.

We see little difference between the smallest possible disruption (Section 3.3) and total runtime randomization (Section 3.6), suggesting Danglot et al.'s [4] correctness attraction will hold more widely than their +1 disruption. Indeed these experiments with integer functions, support the view that software is not fragile [18].

The average depth, rather than size, is critical. With nesting deeper than 5–7 levels it becomes impossible to see the effect of most individual crossovers or mutations and the fitness landscape becomes increasingly flat and evolution harder. This suggests either the need to limit the depth of crossover and mutation or the need to move fitness testing from the root node to closer to the genetic changes [12].

Acknowledgments

This work was inspired by conversations at Dagstuhl Seminar 18052 on Genetic Improvement of Software [21]. I am grateful for the assistance of anonymous reviewers. Supported by the Meta OOPS project. GPQuick code is available in http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/gp-code/GPinc.tar.gz

REFERENCES

[1] Kelly Androutsopoulos et al. 2014. An Analysis of the Relationship between Conditional Entropy and Failed Error Propagation in Software Testing. In ICSE 2014. ACM, Hyderabad, India, 573–583.
[2] Alexander Brownlee et al. 2020. Injecting Shortcuts for Faster Running Java Code. In *CEC*. IEEE.
[3] Jason M. Daida et al. 2005. Visualizing Tree Structures in Genetic Programming. *GP & EM* 6, 1 (March 2005), 79–110.
[4] Benjamin Danglot et al. 2018. Correctness attraction: a study of stability of software behavior under runtime perturbation. *Empirical Software Engineering* 23, 4 (1 Aug. 2018), 2086–2119.
[5] 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.
[6] John R. Koza. 1992. *Genetic Programming*. MIT Press.
[7] W. B. Langdon. 2000. Quadratic Bloat in Genetic Programming. In *GECCO-2000*. 451–458.
[8] W. B. Langdon. 2020. Multi-threaded Memory Efficient Crossover in C++ for Generational Genetic Programming. *SIGEVOLution* 13, 3 (Oct. 2020), 2–4.
[9] W. B. Langdon. 2021. Fitness First. In *GPTP* Wolfgang Banzhaf et al. (Eds.). Springer, 143–164.
[10] W. B. Langdon. 2021. Fitness First and Fatherless Crossover. In *GECCO Companion*. ACM, 253–254.
[11] W. B. Langdon. 2021. Incremental Evaluation in Genetic Programming. In *EuroGP 2021 (LNCS, Vol. 12691)*, Ting Hu et al. (Eds.). Springer Verlag, 229–246.
[12] W. B. Langdon. 2022. Evolving Open Complexity. *SIGEVOlution* 15, 1 (March 2022).
[13] W. B. Langdon. 2022. Failed Disruption Propagation in Integer Genetic Programming. ArXiv 2204.13997.
[14] W. B. Langdon et al. 1999. The Evolution of Size and Shape. In *Advances in Genetic Programming 3*, Lee Spector et al. (Eds.). MIT Press, Chapter 8, 163–190.
[15] W. B. Langdon et al. 2021. Dissipative Polynomials. In *5th Workshop on Landscape-Aware Heuristic Search (GECCO 2021 Companion)*, Nadarajen Veerapen et al. (Eds.). ACM, 1683–1691.
[16] W. B. Langdon et al. 2022. Measuring Failed Disruption Propagation in Genetic Programming. In *GECCO '22*. ACM 964–972.
[17] W. B. Langdon and Wolfgang Banzhaf. 2022. Long-Term Evolution Experiment with Genetic Programming. *Artificial Life* 28, 2 (2022) 173–204.
[18] W. B. Langdon and Justyna Petke. 2015. Software is Not Fragile. In *Complex Systems Digital Campus E-conference, CS-DC'15 (Proceedings in Complexity)*. Springer, 203–211.
[19] W. B. Langdon and R. Poli. 1997. Fitness Causes Bloat. In *Soft Computing in Engineering Design and Manufacturing*. Springer, 13–22.
[20] Edward N. Lorenz. 1993. The Butterfly Effect. In *The Essence of Chaos*. University of Washington Press, Seattle, USA, Chapter Appendix 1, 181–184.
[21] Justyna Petke et al. 2018. Genetic Improvement of Software: Report from Dagstuhl Seminar 18052. *Dagstuhl Reports* 8, 1 (23 July 2018), 158–182.
[22] Justyna Petke et al. 2021. Software Robustness: A Survey, a Theory, and Some Prospects. In *ESEC/FSE 2021, Ideas, Visions and Reflections*. ACM, 1475–1478.
[23] Jeffrey M. Voas. 1992. PIE: a dynamic failure-based technique. *IEEE TSE* 18, 8 (Aug 1992), 717–727.