Dissipative Polynomials

William B. Langdon, Justyna Petke and David Clark
W.Langdon@cs.ucl.ac.uk,j.petke@ucl.ac.uk,david.clark@ucl.ac.uk
Department of Computer Science, University College London
London, UK

ABSTRACT

Limited precision floating point computer implementations of large polynomial arithmetic expressions are nonlinear and dissipative. They are not reversible (irreversible, lack conservation), lose information, and so are robust to perturbations (anti-fragile) and resilient to fluctuations. This gives a largely stable locally flat evolutionary neutral fitness search landscape. Thus even with a large number of test cases, both large and small changes deep within software typically have no effect and are invisible externally. Shallow mutations are easier to detect but their RMS error need not be simple.

KEYWORDS

genetic programming, information loss, information funnels, entropy, evolvability, mutational robustness, neutral networks, SBSE, software robustness, Correctness Attraction, diversity, software testing, theory of bloat, introns

ACM Reference Format:
William B. Langdon, Justyna Petke and David Clark. 2021. Dissipative Polynomials. In *2021 Genetic and Evolutionary Computation Conference Companion (GECCO '21 Companion), July 10–14, 2021, Lille, France.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3449726.3463147

1 INTRODUCTION

Large arithmetic expressions are resilient to change (e.g. bugs).

We sample uniformly the space of large polynomials using addition and multiplication and show changes (bugs) usually do not impact expressions values. Further even testing as many as a thousand test points (uniformly selected in the range -1.0 to +1.0), the disruption caused by changes to the functions on average penetrates only about 100 nested levels and so they are often invisible outside the expression. With fewer tests (i.e. a weaker test oracle), changes impact fewer levels. However, with such a large number of tests, chance disruption close to the outermost part of the expression (the root node) may indeed have a sizable effect. Note the effect of the inserted change/bug progressively fails to propagate through the expression since both arithmetic operators (+ and $\times$) are dissipative (irreversible, lose information) in practice. (See also [2, 10–14, 18, 20–22, 26].)

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
GECCO '21 Companion, July 10–14, 2021, Lille, France
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8351-6/21/07...$15.00
https://doi.org/10.1145/3449726.3463147

In an idealised computer with infinite precision, sometimes small changes might be visible. However if the injected perturbation is far from the top of the expression, real effects, such as floating point precision and rounding error, may smooth away changes.

For evolutionary computing [8, 27], this means large complex expressions are robust and present a smooth fitness landscape where many mutations have no measurable effect or their impact is only seen on some test cases. Whilst we deal exclusively with arithmetic expressions, there is growing evidence that this is true of programming in general [19], [26], [17], [24], [31], [4], [7].

In the case of programming typically there are side-effects. Nonetheless it appears that it is common for there to be information funnels, whereby large amounts of information inside the program are reduced into a small amount visible externally. If we view each nested operation or function call in a program as being analogous to a level in our nested polynomials, then we can view huge software stacks, which are common in modern computer systems, as being somewhat similar to the large polynomials we investigate in the following sections. Notice one of the reasons why software engineers try to test small parts of huge software systems in isolation (unit testing) is the difficulty of seeing externally the impact of deeply nested errors or bugs.

Sections 3 and 4 give more details on sampling all possible expressions and changes to them. In the experimental section (Section 5) addition and multiplication are performed on ordinary 32 bit floating point numbers. We create large arithmetic expressions, make small changes to them, and trace the impact of the change. In most cases, the impact dies away before it can affect anything outside the expression. Section 6 analyses in more detail the dissipation of one of the larger changes sampled. In Section 7 we conclude information loss makes software robust but this makes software engineering, e.g. bug fixing, harder and makes large evolutionary computing search landscapes smooth and so difficult to search.

2 WHY SOME CHANGES ARE INVISIBLE

In genetic programming [8, 27] the idea of useless bloated code is well known. Indeed the term intron [30],[1],[29] is often used to refer to code that has no, or little, impact on the program's output(s). For example, a subtree "ored" with true has no effect as the OR function will always return true regardless of the intron. Similarly multiplying by 0 always gives 0, so MUL's other argument has no effect and can also be said to be in an intron. (There are automatic expression simplification tools which will spot and remove obvious introns.) Traditional introns can explain in evolved code many examples of large expressions being robust to changes. However the phenomenon is general.

If a leaf 0.892 is changed to 0.992, it is as if an error of 0.1 is injected into the expression at that point. In a small expression, this might be easily observed. In a large expression the 0.1 error

is transformed by each function it passes through. In general the error may become bigger or smaller. Even under ideal conditions, floating point arithmetic loses about half a bit of precision at each operation. So disruption is progressively suppressed. As the expressions are hierarchical, once disruption on a test case is lost (i.e. an internal function yields the same answer before and after the change) it cannot be reintroduced higher in the tree. (Note the number of disrupted test cases falls monotonically.) This continued interference of finite computing, means the impact of even large errors can be totally lost in large expressions.

3 UNIFORMLY SAMPLING LARGE ARITHMETIC EXPRESSIONS

All the experiments sample uniformly the space of expressions with 12 500 arithmetic operators. As both operators have two inputs, the expressions are binary trees with 12 500 internal nodes and 12 501 leafs (or external nodes). For a particular input x, the value of the whole expression is the value calculated by the operator that is the root node of the tree. (Figure 1 shows an example subexpression. Note in all our pictures of polynomials as trees, the result returning part, the root, is at the top. Figure 2 shows the first example expression.)

We chose a tree size of 25 001 (internal + external), since such trees on average have a depth of close to $\sqrt{2\pi|\text{size}|} \approx 400$ (Flajolet and Oldyzko [5]) and our earlier work with evolved trees showed on average typically the impact of mutations was lost after traversing about 100 functions [16]. 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.)

The leafs are also sampled. With a probability of 50% the input x is chosen. The other leafs are chosen uniformly from 250 constant values chosen at random without replacement from the 2001 multiples of 0.001 between -1.0 and +1.0. By random chance, none of the special values -1.0, 0, or +1.0 are included.

4 SAMPLING CHANGES

A site for each change is selected uniformly from each large expression. The subexpression at that location is removed and replaced by another subexpression. The inserted subexpression is similarly chosen uniformly from a large expression of the same size (i.e. 25 001 nodes). (Cf. Koza's subtree crossover [8].)

Table 1 describes the random changes. They are plotted in Figure 3. Notice (column 3 in Table 1), by chance, change 9 is closest to its root node (depth 47) and is the only example where a change is visible on any of our 1001 test cases.

5 EXPERIMENT: DISSIPATION OF CHANGE

Figure 4 shows the ten large randomly chosen polynomials.

Figure 5 confirms the monotonic fall in disruption of test cases as we move away from the disruption. (I.e., as we move up the tree towards the root node.) Figure 6 shows the same thing, but instead of counting the number of test cases which are not identical, Figure 6 plots the average (root mean square, RMS) difference before and after the change in the values inside each of the large expressions. Figure 6 shows, as expected, RMS differences can rise as well as fall

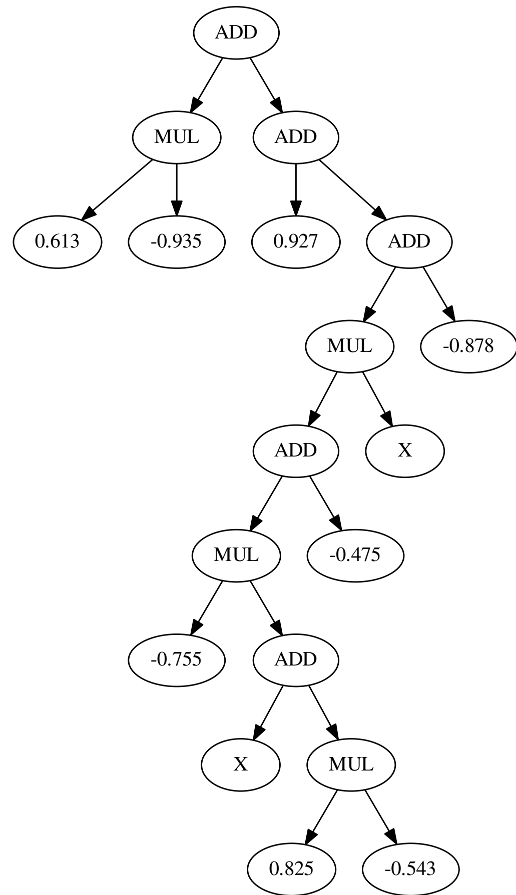

Figure 1: Example subexpression $(0.613 \times -0.935) + 0.927 + (-0.755 \times (x + (0.825 \times -0.543)) - 0.475)x - 0.878$ **as a tree (fun 0 from Table 1). The value of the expression is given by the root node, here ADD, at the top. And plotted as an inverted parabolic red line with + in Figure 3.**

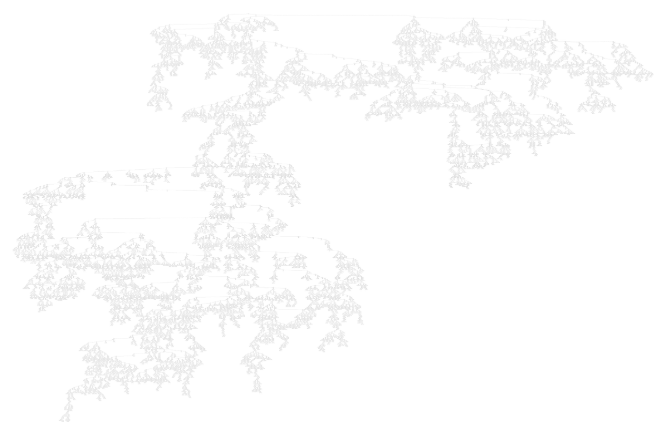

Figure 2: Expression 0 presented as a binary tree of 25 001 nodes (depth 383). Root node at top.

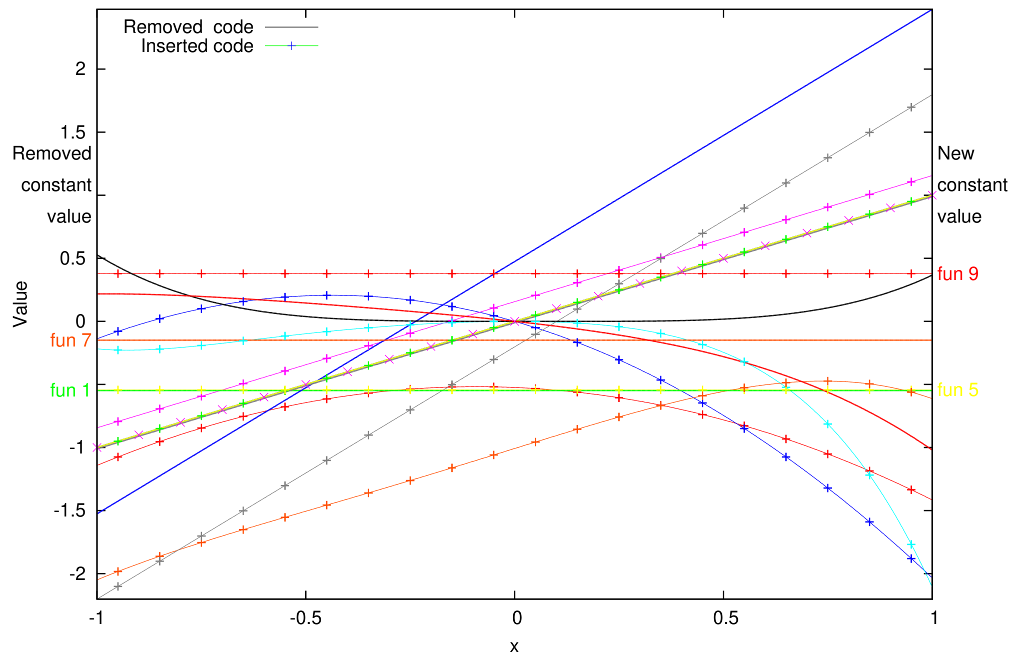

Figure 3: Ten pairs of changes plotted as functions of x, see Table 1. Inserted subexpressions are plotted with lines and crosses. Horizontal lines indicate constants. Labels on the left margin indicates constant values that are removed. Labels on the right, constants that are inserted. In three cases x is removed and in two, x is inserted. These are plotted on top of each other along the diagonal. E.g. in fun 7 the constant -0.149 is replaced by a randomly chosen non-linear function.

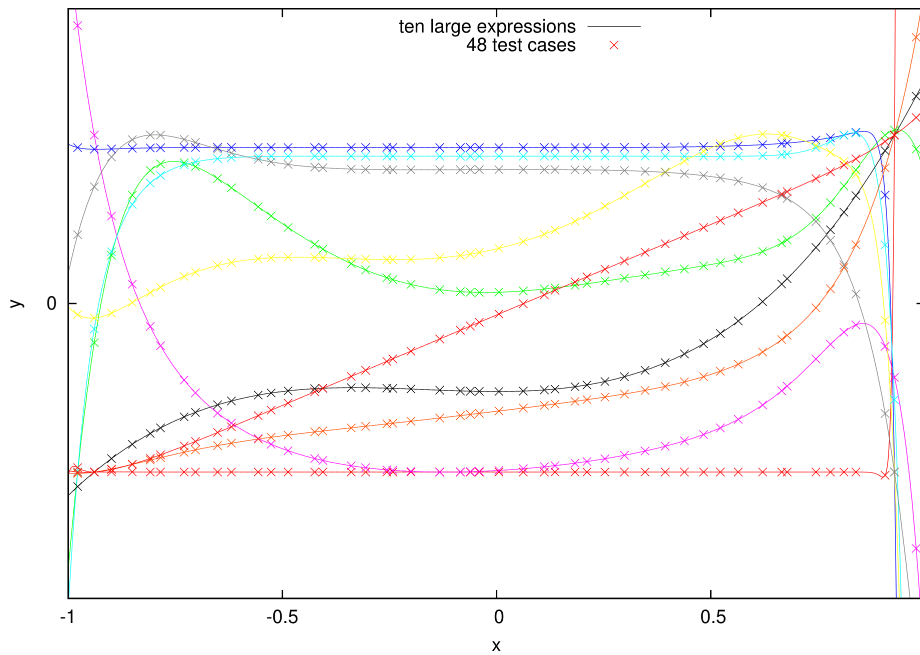

Figure 4: Ten large floating point functions. Vertical axis has been linearly rescaled to plot very different output ranges on the same axis.

Test	Depth Max	Depth Mean	Change depth	size	and function
0	383	162.9	163	1	X
replaced by fun 0				19	(ADD (MUL 0.613 -0.935) (ADD 0.927 (ADD (MUL (ADD (MUL -0.755 (ADD X (MUL 0.825 -0.543))) -0.475) X) -0.878)))
1	377	206.3	325	1	-0.549
replaced by fun 1				1	X
2	305	161.8	190	5	(ADD X (ADD 0.474 X))
replaced by fun 2				9	(MUL (ADD X X) (ADD -0.474 (MUL X -0.543)))
3	376	180.6	148	1	X
replaced by fun 3				7	(ADD (ADD 0.265 X) (MUL 0.837 -0.13))
4	491	253.1	298	1	X
replaced by fun 4				67	(MUL (ADD (ADD (MUL 0.581 (MUL (ADD X 0.837) (ADD (ADD (MUL 0.255 -0.622) X) (ADD X 0.113)))) (MUL X -0.801)) 0.965) (MUL X (MUL (ADD (MUL 0.758 (MUL (ADD X (MUL (MUL -0.07 (MUL (ADD (ADD (MUL (MUL -0.399 X) -0.285) X) 0.185) X)) (MUL (MUL 0.255 (ADD (MUL (MUL 0.14 (ADD X -0.015)) -0.619) (ADD X -0.106))) (ADD (ADD X X) X)))) X)) X) -0.546)))
5	417	192.1	167	1	X
replaced by fun 5				1	-0.543
6	504	229.2	224	49	(ADD (MUL 0.653 (MUL X (MUL (MUL X (ADD X (MUL (ADD X (ADD 0.305 (MUL -0.247 X))) -0.415))) X))) (MUL (MUL (ADD (MUL X (MUL 0.289 0.263)) (ADD X -0.147)) (MUL (MUL X (MUL (ADD (ADD -0.272 -0.67) X) (MUL 0.379 0.176))) (MUL X 0.038))) (MUL 0.662 X)))
replaced by fun 6				1	X
7	345	181.1	175	1	-0.149
replaced by fun 7				37	(ADD X (ADD (MUL -0.756 (ADD (ADD 0.667 0.558) (MUL (MUL 0.411 (MUL X (ADD (ADD (MUL 0.243 0.243) 0.752) X))) (ADD (ADD (ADD (MUL X X) (MUL 0.443 X)) (MUL -0.294 (MUL X 0.892))) -0.106)))) -0.082))
8	332	160.6	142	3	(ADD -0.011 X)
replaced by fun 8				7	(ADD (MUL -0.636 0.318) (ADD X X))
9	390	189.3	47	77	(MUL (MUL (MUL -0.546 0.371) 0.704) (ADD (MUL (ADD X (ADD (MUL (MUL (ADD -0.876 (ADD (MUL (MUL 0.979 (MUL 0.474 0.522)) X) X)) X) X) (MUL X (ADD X (ADD 0.217 (ADD -0.27 X)))))) (ADD 0.558 X)) (ADD (MUL (MUL (MUL X -0.889) (MUL (MUL (ADD (MUL -0.718 (ADD (ADD X 0.593) X)) X) (ADD X 0.309)) (MUL 0.048 (MUL X (MUL 0.879 (MUL -0.831 (ADD X (MUL 0.185 (ADD 0.912 X))))))))) 0.146) (ADD X X))))
replaced by fun 9				1	0.379

Table 1: Ten changes. Left: maximum and average depths of ten uniformly selected polynomials with 25 001 nodes (Section 3). Right: uniformly selected changes (Section 4). Column 4 shows the depth of the uniformly chosen change site, whilst column 5 gives the size of the removed subexpression and its replacement and the rightmost column (6) shows them as prefix (lisp like) expressions. (See also Figures 1 and 3.)

but where the change is deep enough, differences also eventually fall to zero.

In nine of ten cases disruption (red subtrees and string of blue nodes in Figure 7) is halted before reaching the root node. Even in the remaining case, fun 9, disruption is rapidly quenched but does not quite reach zero before encountering the limit of the polynomial.

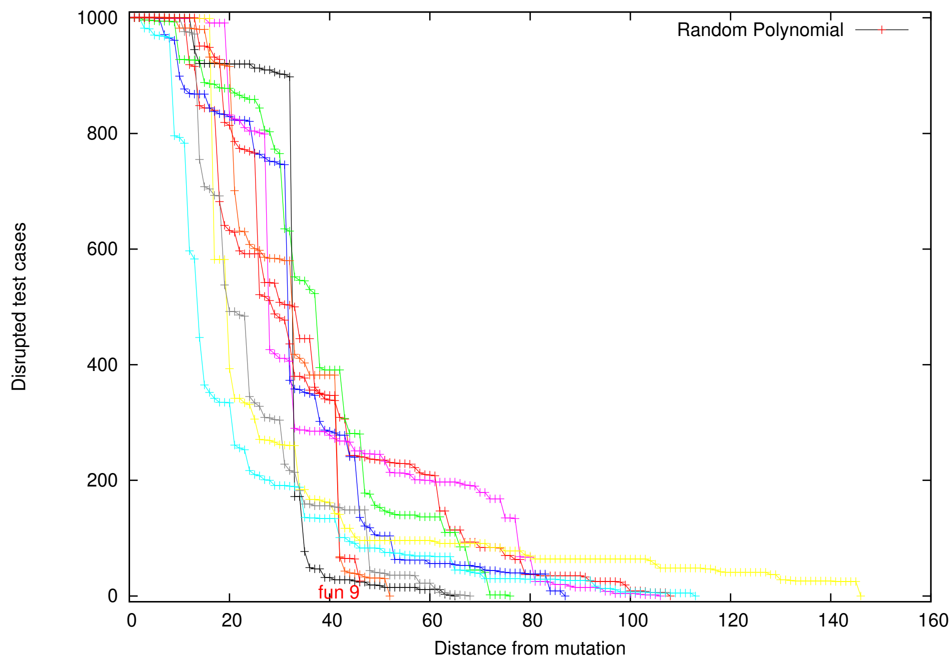

Figure 5: Fall in impact of ten changes with distance from disruption. As expected the fall in test case failures is monotonic. Only fun 9 does not reach zero (26 of 1001 test cases not identical at root node). Colours are the same as in Figure 3, etc.

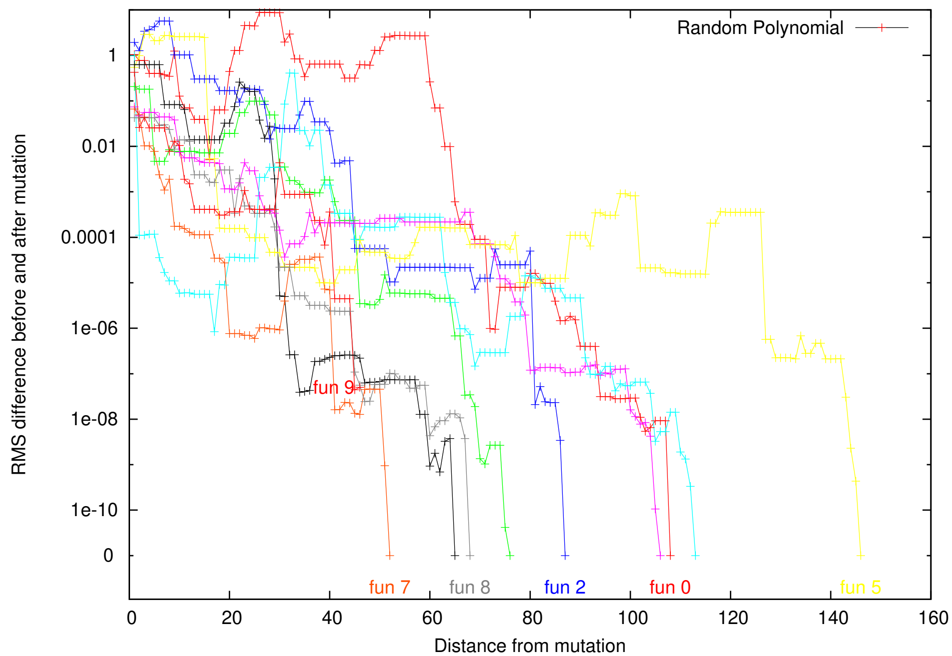

Figure 6: Impact of ten changes against distance from change location. (Impact measured by root mean squared difference on 1001 test cases.) Only fun 9 does not reach zero (RMS difference $5\ 10^{-8}$ at root node). Colours are the same as in Figure 3, etc. Note non-linear vertical scale.

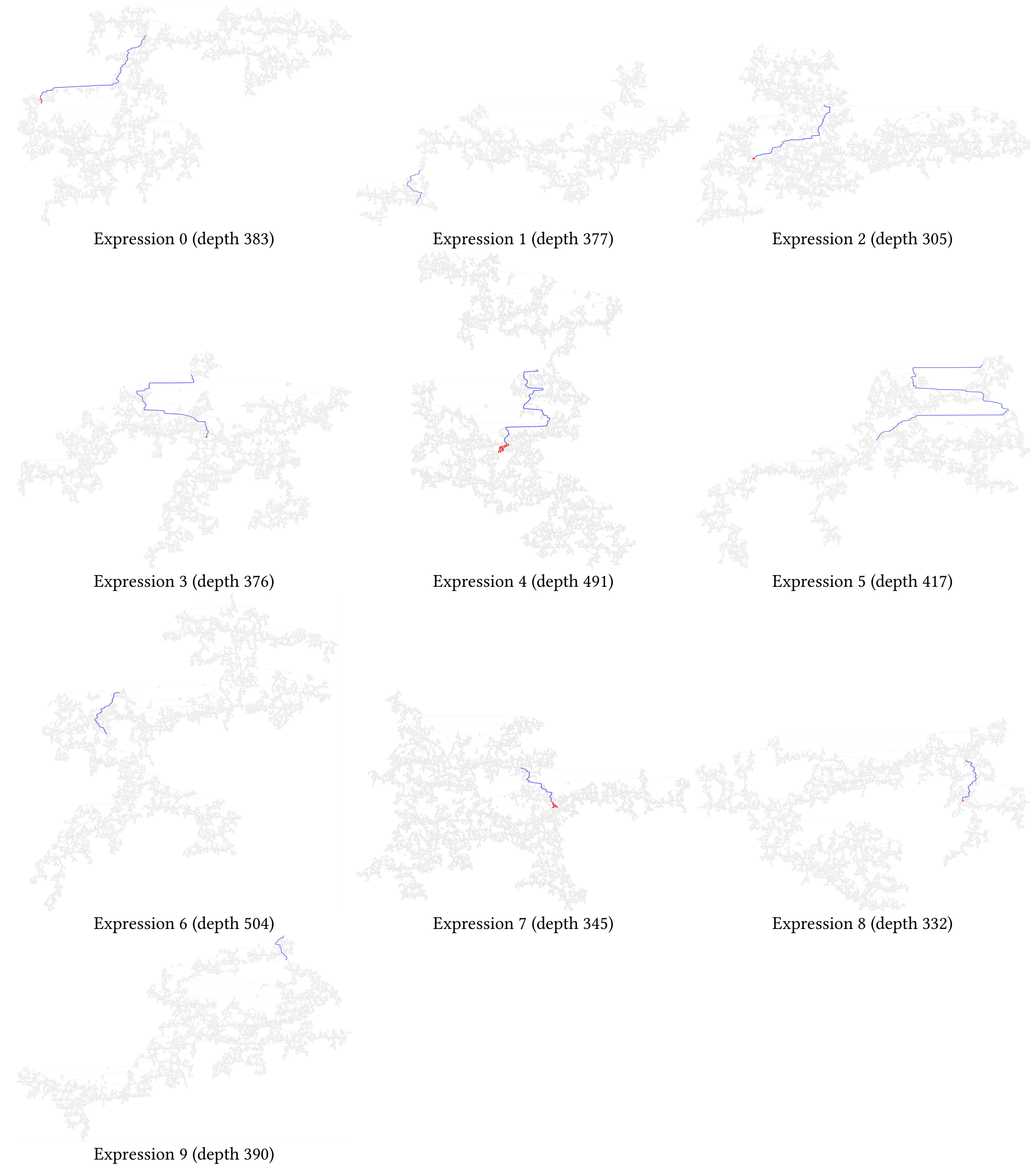

Figure 7: Expressions 0–9 presented as a binary trees of 25 001 nodes. Root nodes at top. Colour indicates disrupted nodes. Red (lowest shaded nodes) shows a new subexpression replacing an earlier subexpression. Blue nodes show subexpressions where at least one test case produces a different internal value as a result of the change. Notice only in polynomial 9 does any part of the disruption reach the root node.

6 EXPLAINING LACK OF IMPACT OF CHANGE 4

Table 1 shows at 67 nodes, change 4 is one of the larger syntactic changes. Indeed the light blue line in Figure 3 shows it also produces a large change in behaviour at the change site. (Change 4 replaces a single x by a large expression which is quadratic in x.) See also Figure 8. At the point of disruption all but one test case are different and the RMS difference is 0.98 (light blue line in Figure 6).

At the disruption point the new polynomial is different from the original at all test points (except $x = 0$). However, notice except for the changed code (red in Figure 8), for the test case $x = 0$ all of the original and the new code must be identical. Therefore at $x = 0$ the new and the original polynomial must have the same value.

The next function up is an addition and the one above that is a multiply. Neither reduces the number of non-identical test cases, however the multiply reduces the average difference from about 1 to about 0.0001. The third function is another addition, which reduces the number of non-identical test cases by 18 (see Figure 9). The next function is a multiply, which further synchronises the new and the original polynomial on an additional test point. The next function is an addition which reduces the RMS difference to zero on a further nine points (see Figure 10). By Figure 11 (top blue node in Figure 8) 205 of 1001 test points are identical.

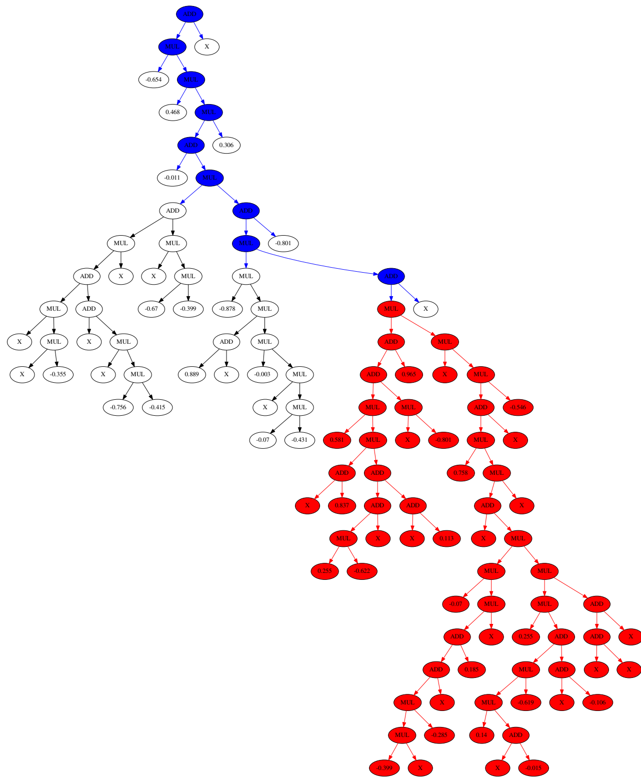

Figure 8: Fragment of change 4. The original leaf X is replaced by the 67 node subexpression (MUL (ADD (ADD (MUL 0.581 (MUL (ADD X 0.837) (ADD (ADD (MUL 0.255 -0.622) X) (ADD X 0.113)))) (MUL X -0.801)) 0.965) (MUL X (MUL (ADD (MUL 0.758 (MUL (ADD X (MUL (MUL -0.07 (MUL (ADD (ADD (MUL (MUL -0.399 X) -0.285) X) 0.185) X)) (MUL (MUL 0.255 (ADD (MUL (MUL 0.14 (ADD X -0.015)) -0.619) (ADD X -0.106))) (ADD (ADD X X) X)))) X)) X) -0.546))) in red. The blue nodes show operations in the original expression where their value on ≥ 796 of 1001 test cases are different before and after change 4. White nodes show fragment of unchanged large expression, shown in full in Figure 7. The value of the new subexpression is given by its top most node, here red MUL and plotted as a function of x in Figure 3 (light blue line). Section 6 explains why disruption stops completely after 113 blue nodes and the red change make no visible external difference.

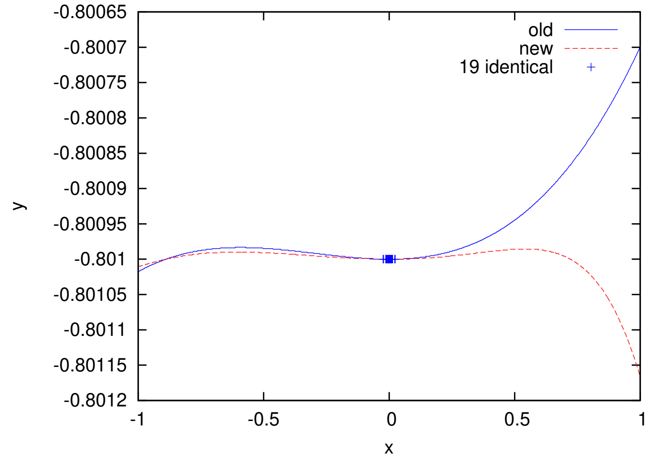

Figure 9: Impact of change 4 at distance three above the change point (ADD -0.801, Figure 8). The new functionality (dashed line) closely follows the original for $x < 0.2$ and indeed at 19 points (+) they are identical.

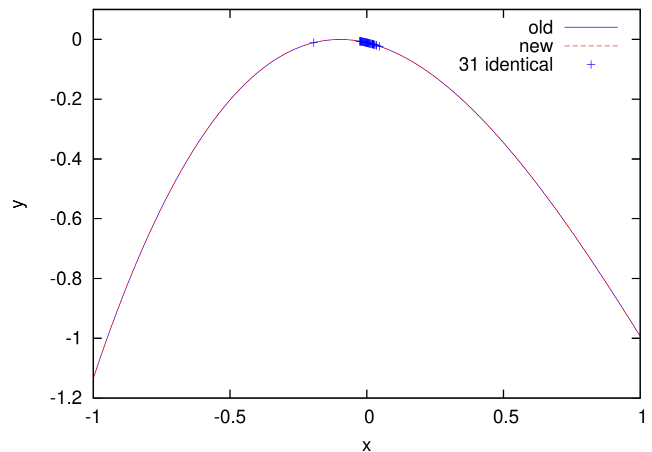

Figure 10: Impact of change 4 at distance five (ADD -0.011). The new functionality (dashed line) closely follows the original and indeed at 31 points (+) they are identical.

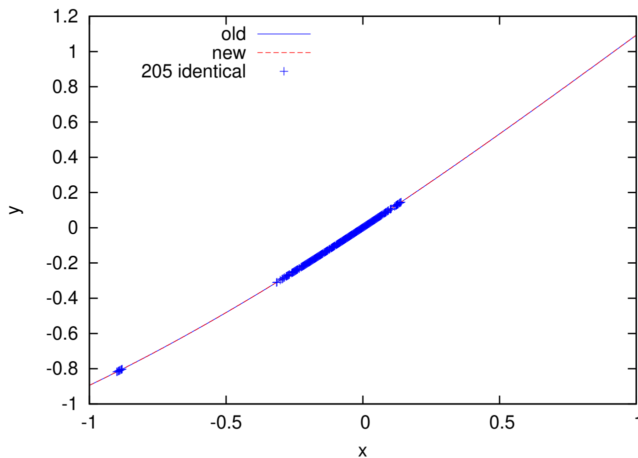

Figure 11: Impact of change 4 at distance nine (ADD X, top of Figure 8). The new functionality (dashed line) closely follows the original and they are identical at 205 points (+).

7 CONCLUSIONS

Almost all computing operations are irreversible. Meaning after they have acted it is impossible to know what the state of the computation was before. For example, adding two registers (r0, r1) and storing the result in another register (r2). We cannot tell from the answer which two numbers were added. E.g. 99+1=100 but so does 98+2, 97+3, and so on[1]. From an information theoretic viewpoint, we can say that addition has taken two values with up to 32 bits of information in each (i.e. $\leq$ 64 bits in total) and produced a 32 bit answer, which can contain at most 32 bits of information[2]. That is, irreversible operations must lose information.

In the case of polynomials, treating them as side effect free trees makes it plain that information can only flow from their leafs towards their root, and once information is lost at any point within the tree, it is gone for good. It cannot be recreated.

A special case of information loss, is software testing [32]. If we view our actual code as being a mixture of perfect code plus an "error", we can analyse the actual code's behaviour by analysing the impact of the error on the information (data) flow of the perfect program. To have any impact, the error needs to be executed, to change the state of the computation and that change has to be propagated to a point where it is visible outside the program (e.g. a print statement). Notice information has to be passed through the computation. Although the information may be stored in memory, in many programs it has at some point to pass along a chain of irreversible information losing computations and as we have seen as that chain gets longer (e.g. the error is in more deeply nested function calls) there is an increasing chance that it will be lost and so the error will not be visible externally.

The upside of this is: the bug has no effect, whilst the glass half empty view is: that testing to find bugs, is more difficult. That is, information loss is inevitable and in general makes complex software resilient or anti-fragile [19], [4], [28], [7], [23], [3], [6], [17].

From an evolutionary computing perspective, the same holds. That is, in the above, if we replace error/bug by mutation or crossover change, we will see that changes made far from the impact point of our genome are liable to have little impact on fitness. Conversely changes near the root node (if we are using trees) or the drive of our robot are likely to have more impact on fitness. It also appears that mutations deep within the tree or controller will need considerably more (possibly exponentially more) fitness testing. Thus bigger trees or larger control structures are liable to have a smoother landscapes with larger plateaus.

[1]Although we have not over written r0 and r1, and so they do contain their original values, we have over written r2, so its early value is now unknown.

[2]Although we have only used standard (32 bit) floating point arithmetic, the same arguments apply to double precision (64 bit) and even 128 bit arithmetic. That is, they too will losing information. We suggest that possibly higher precision operations will tend to be less dissipative and consequently more of them, corresponding to more deeply nested function calls, will be needed to give the same concealment of changes. Elsewhere [9] we suggest that the number of nested functions needed to conceal changes tends to increase only slowly, as O(log n), with the number (n) of tests. Perhaps we will see a similar O(log n) scaling with number (n) of bits of precision in the floating point resolution. However we have not proved this.

Acknowledgments

This work was inspired by conversations at Dagstuhl Seminar 18052 on Genetic Improvement of Software [25].

I am grateful for the assistance of the anonymous reviewers.

Funded by EPSRC grant EP/P005888/1.

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

REFERENCES

[1] 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

[2] Bobby R. Bruce et al. 2019. Approximate Oracles and Synergy in Software Energy Search Spaces. *IEEE Transactions on Software Engineering* 45, 11 (Nov. 2019), 1150–1169. http://dx.doi.org/10.1109/TSE.2018.2827066

[3] David Clark et al. 2020. Software Robustness: A Survey, a Theory, and Some Prospects. Presented at Facebook Testing and Verification Symposium 2020.

[4] 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 (August 2018), 2086–2119. http://dx.doi.org/10.1007/s10664-017-9571-8

[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. https://doi.org/10.1016/0022-0000(82)90004-6

[6] Giovani Guizzo et al. 2021. Enhancing Genetic Improvement of Software with Regression Test Selection. In *Proceedings of the International Conference on Software Engineering, ICSE 2021*, Arie van Deursen et al. (Eds.). IEEE. http://dx.doi.org/10.1109/ICSE43902.2021.00120 Winner ACM SIGSOFT Distinguished Artifact Award.

[7] Nicolas Harrand et al. 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

[8] 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

[9] W. B. Langdon. [n.d.]. Genetic Programming Convergence. *Genetic Programming and Evolvable Machines* ([n. d.]). http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/langdon_GPEM_gpconv.pdf

[10] W. B. Langdon. 2003. Convergence of Program Fitness Landscapes. In *Genetic and Evolutionary Computation – GECCO-2003 (LNCS, Vol. 2724)*, E. Cantú-Paz et al. (Eds.). Springer-Verlag, Chicago, 1702–1714. http://dx.doi.org/10.1007/3-540-45110-2_63

[11] W. B. Langdon. 2003. The distribution of Reversible Functions is Normal. In *Genetic Programming Theory and Practice*, Rick L. Riolo and Bill Worzel (Eds.). Kluwer, Chapter 11, 173–187. http://dx.doi.org/10.1007/978-1-4419-8983-3_11

[12] W. B. Langdon. 2005. The Distribution of Amorphous Computer Outputs. In *The Grand Challenge in Non-Classical Computation: International Workshop*, Susan

Stepney and Stephen Emmott (Eds.). York, UK. http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/grand_2005.pdf
[13] W. B. Langdon. 2009. Scaling of Program Functionality. *Genetic Programming and Evolvable Machines* 10, 1 (March 2009), 5–36. http://dx.doi.org/10.1007/s10710-008-9065-y
[14] W. B. Langdon. 2012. Genetic Improvement of Programs. In *18th International Conference on Soft Computing, MENDEL 2012* (2nd ed.), Radomil Matousek (Ed.). Brno University of Technology, Brno, Czech Republic. http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/Langdon_2012_mendel.pdf Invited keynote.
[15] 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
[16] 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 et al. (Eds.). Springer Verlag, Virtual Event, 229–246. http://dx.doi.org/10.1007/978-3-030-72812-0_15
[17] W. B. Langdon et al. [n.d.]. ([n. d.]). Submitted.
[18] William B. Langdon and Mark Harman. 2016. Fitness Landscape of the Triangle Program. In *PPSN-2016 Workshop on Landscape-Aware Heuristic Search*, Nadarajen Veerapen and Gabriela Ochoa (Eds.). Edinburgh. http://www.cs.ucl.ac.uk/fileadmin/UCL-CS/research/Research_Notes/rn1605.pdf Also available as UCL RN/16/05.
[19] 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 et al. (Eds.). Springer, 203–211. http://dx.doi.org/10.1007/978-3-319-45901-1_24 Invited talk.
[20] W. B. Langdon and R. Poli. 2006. The Halting Probability in von Neumann Architectures. In *Proceedings of the 9th European Conference on Genetic Programming (Lecture Notes in Computer Science, Vol. 3905)*, Pierre Collet et al. (Eds.). Springer, Budapest, Hungary, 225–237. http://dx.doi.org/10.1007/11729976_20
[21] William B. Langdon and Riccardo Poli. 2006. On Turing complete T7 and MISC F–4 program fitness landscapes. In *Theory of Evolutionary Algorithms (Dagstuhl Seminar Proceedings, 06061)*, Dirk V. Arnold et al. (Eds.). Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, Dagstuhl, Germany. http://drops.dagstuhl.de/opus/volltexte/2006/595 <http://drops.dagstuhl.de/opus/volltexte/2006/595> [date of citation: 2006-01-01].
[22] William B. Langdon and Riccardo Poli. 2008. Mapping Non-conventional Extensions of Genetic Programming. *Natural Computing* 7 (March 2008), 21–43. Issue 1. http://dx.doi.org/10.1007/s11047-007-9044-x Invited contribution to special issue on Unconventional computing.
[23] Mingyi Lim et al. 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
[24] 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
[25] Justyna Petke et al. 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
[26] Justyna Petke et al. 2019. A Survey of Genetic Improvement Search Spaces. In *7th edition of GI @ GECCO 2019*, Brad Alexander et al. (Eds.). ACM, Prague, Czech Republic, 1715–1721. http://dx.doi.org/10.1145/3319619.3326870
[27] Riccardo Poli et al. 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).
[28] Joseph Renzullo et al. 2018. Neutrality and Epistasis in Program Space. In *GI-2018, ICSE workshops proceedings*, Justyna Petke et al. (Eds.). ACM, Gothenburg, Sweden, 1–8. http://dx.doi.org/10.1145/3194810.3194812 Best Presentation Award.
[29] Craig W. Reynolds. 1994. Evolution of Corridor Following Behavior in a Noisy World. In *Simulation of Adaptive Behaviour (SAB-94)*, David Cliff et al. (Eds.). MIT Press, Brighton, UK, 402–410. http://www.red3d.com/cwr/papers/1994/sab94.pdf
[30] 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
[31] 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.
[32] 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