EuroGP-2021. Ting Hu, Nuno Lourenco, Eric Medvet *Eds.*, LNCS 12691, 229–246, Internet. 7-9 April 2021. Springer. doi:10.1007/978-3-030-72812-0_15 *Preprint*

Incremental Evaluation in Genetic Programming

William B. Langdon

W.Langdon@cs.ucl.ac.uk CREST, Department of Computer Science, UCL, Gower Street, London, WC1E 6BT, UK

Abstract. Often GP evolves side effect free trees. These pure functional expressions can be evaluated in any order. In particular they can be interpreted from the genetic modification point outwards. Incremental evaluation exploits the fact that: in highly evolved children the semantic difference between child and parent falls with distance from the syntactic disruption (e.g. crossover point) and can reach zero before the whole child has been interpreted. If so, its fitness is identical to its parent (mum).

Considerable savings in bloated binary tree GP runs are given by exploiting population convergence with existing GPquick data structures, leading to near linear O(gens) runtime. With multi-threading and SIMD AVX parallel computing a 16 core desktop can deliver the equivalent of 571 billion GP operations per second, 571 giga GPop/s.

GP convergence is viewed via information theory as evolving a smooth landscape and software plasticity. Which gives rise to functional resilience to source code changes. On average a mixture of 100 +, -, × and (protected) ÷ tree nodes remove test case effectiveness at exposing changes and so fail to propagate crossover infected errors.

Keywords parallel computing, mutational robustness, antifragile correctness attraction, PIE, SBSE, software resilience, entropy loss, theory

1 Background: Genetic Programming Evolving Functions

Most GP problems, such as symbolic regression or classification, require the evolution of a pure function. I.e., a function without side effects. Exceptions include: problems with state, such as the truck-backer-upper problem [1] or Santa Fe Trail problem [1,2], which involve an agent moving in an environment with memory, where state is embedded in the program itself [3,4] or where genetic programming is applied to existing programs (as in genetic improvement [5],[6,7,8,9,10]). Typically evolution decides the number and nature of the evolved function's inputs as well as the contents of the function itself. Essentially the idea is we do not know what we want but we can recognise a good function when GP finds one. We give evolution some way of recognising better functions from poorer ones by automatically allocating each function a fitness score. (Although fitness can be assigned manually [11].) Typically, in both regression and classification, the fitness function uses multiple examples where all the inputs and the example's expected output are known. (Backer describes dealing with missing input values [12].) An evolved function's fitness value is given by calling it with each

example's inputs and comparing its return value with the expected value. Typically the differences between the function's return value and the expected answer for several test cases are combined into a single scalar fitness value. (Although multiple objective approaches are increasingly popular [13, Ch. 9].) For classification problems, we may use the area under the ROC curve [14], whilst for the continuous domain regression problems, the mean absolute error remains popular. Notice, unlike Gathercole's DSS [15] and Lexicase selection [16] or other dynamic choice of test data, e.g. [17,6], typically GP systems use all of a fixed training set of examples throughout evolution. Often a holdout set is needed to estimate the degree of overfitting [13, p140]. In most cases almost all GP's computational cost comes from running the fitness function [1]. As the evolved functions become deeper, information in their leafs is combined, eventually all of it being channeled through the limited capacity of the root node. On the way to the root information is progressively lost. Thus deep functions are resilient to changes near their leafs this, together with high selection pressure, leads to converged GP populations. We show that this can be exploited to considerably reduce fitness evaluation times and hence GP run time.

We follow common tree GP practice by representing each evolved individual as a separate tree composed of nested binary functions (ADD, SUB, MUL and DIV[1]). Each generation a new population of trees is created by recombining two parent trees from the previous generation to create a new population of trees. Koza et al. [19, pages 1044-1045] [20] showed it is not necessary to store all of both populations simultaneously.

We retain GPquick's [21,22] linear prefix trees [23], although Simon Handley [24] demonstrated it is possible to store the complete evolving GP in a single directed acyclic graph (DAG). (See also Nic McPhee's Sutherland [25].) For pure functions, at the expense of memory, caches of partial results can be embedded in the DAG [26]. When a new individual is created it can be evaluated using the partial results of the subtrees from which its parents are composed. In the case of sub-tree mutation, the new random subtree (typically small) must be evaluated. But then and for subtree crossover, the rest of the individual can be evaluated, using partial results stored in the DAG, requiring only evaluation from the mutation or crossover point all the way to the root node. Ignoring the costs of accessing and maintaining the DAG memory structure, the cost of fitness evaluation of the new individual, is the product of the number of fitness cases (assumed fixed) and the height of the tree. The height of large GP trees is typically $\approx \sqrt{2\pi|\text{size}|}$ [27, page 256] [28] [29] (see Figure 7). I.e. considerably less than typical implementations, which evaluate the whole tree and so scale O(size) rather than O(size$^{0.5}$). However the complexity of implementation, the difficulty of efficient operation on multi-core CPUs, and limited memory cache, mean DAGs are not common in GP. In principle, exploiting GP convergence allows fitness evaluation to scale independently of tree size O(1), although it appears this is only approached for humongously large trees, Figure 1.

[1] (DIV X Y) is protected division which returns 1.0 if Y=0 and X/Y otherwise. Ni et al. [18] propose the analytic quotient instead of DIV.

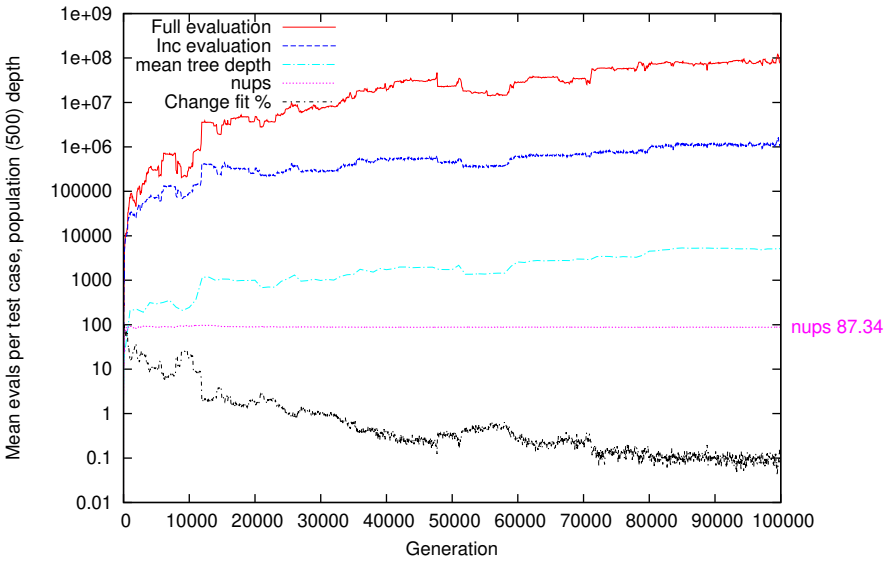

Fig. 1. Convergence of sextic polynomial. Black % parent's $\neq$ child's fitness. After gen 800 most children have identical fitness to mum, and on average (nups, dotted line) incremental evaluation evaluates subtrees of depth ≈100. Max saving in eval ops on traditional (top red v. dashed blue) is 100 fold. Note log scale.

Retaining GPquick's separate linear prefix trees suggests our approach of incremental evaluation could be used with linear postfix trees common in parallel GPs running on graphics hardware accelerators, e.g. GPUs [30,31].

2 Incremental Evaluation: How Does it Work

Since there are no side effects, the functional expressions can be evaluated in any order. In GP it is common to evaluate them recursively from the root down (see Figure 2 left) for the first test case, then the second test case and so on until all the test cases have been run. However it is possible to start from the leafs and work to the root (see Figure 2 right). Particularly when using parallel hardware [32,33,22], it is possible to evaluate each node in the tree on all the test cases, generating a vector of sub-results and propagate these vectors (rather than single scalars) through the tree. Indeed combinations between these extremes are also valid [34]. The results are identical.

When running evolution for a long time with small populations, evolution tends to converge [35]. In GPs this can manifest itself with many different large (and hence expensive to evaluate) individuals producing children with identical fitness [33,29]. Initially it was assumed, at least in the continuous domain, that this was due to large amounts of introns [36,37,38], such as multiplication by zero, which would mean MUL's other subtree had no effect. However, although (SUB X X) can readily produce zero and although the fraction of zeros varies

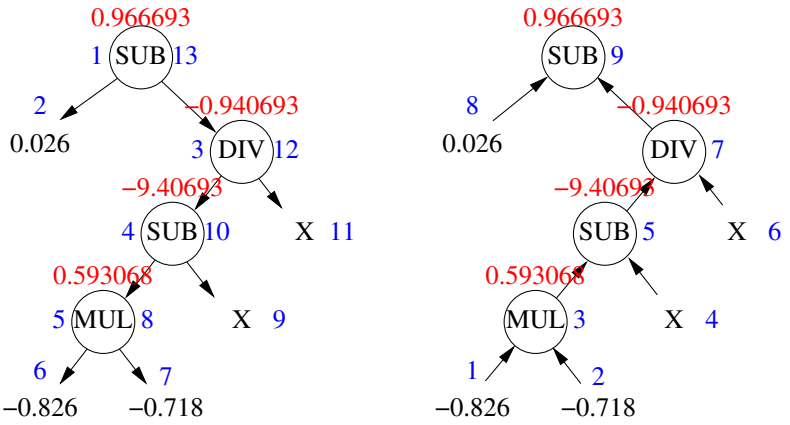

Fig. 2. Left: Conventional top-down recursive evaluation of (SUB 0.026 (DIV (SUB (MUL -0.826 -0.718) X) X)). X=10. Blue integers indicate evaluation order, red floats are node return values. Right: an alternative ordering, starting with leaf -0.826 and working to root node. Both return exactly the same answer.

Table 1. Percentage of functions in evolved trees in ten runs where it gives the same value (0.0, 1.0 or another constant) on all 48 test cases. Data from generation 1000 in Figure 3.

Constant	1	2	3	4	5	6	7	8	9	10	mean
0.0	0.73	1.32	0.23	0.12	0.16	0.10	4.36	2.70	1.18	1.07	1.20 %
1.0	0.15	1.47	0.01	0.00	0.00	1.81	1.87	1.36	1.83	1.90	1.04 %
other	9.13	1.11	17.20	9.33	5.21	16.10	6.89	10.58	12.37	4.91	9.28 %

between runs (see Figure 3 and Table 1), on average only about one percent of tree nodes evaluate to zero for all fitness cases[2].

Since the location of GP genetic operations are typically chosen at random from all the possible locations in the tree, they tend to be far from the tree's root. Although, Koza in [1] defined a small bias away from choosing leafs as crossover points (which we do not use in these experiments), it makes only a small difference. Effectively moving the average crossover point from near the leafs to near the outer most functions. I.e. only one level closer to the root. That is, we can view each child as being the same as the parent (for simplicity called mum) from which it inherits its root node, plus a small disruptive subtree inherited from the other parent (dad). Obviously if the dad subtree is identical to the subtree it replaces from mum, the whole offspring tree is identical to the mum tree and has identical fitness.

For near converged populations of trees of any size, a first optimisation could be to compare the child with its mum. If they are identical, the child's fitness is identical to its mum's fitness and the child does not have to be evaluated. Notice we do not even have to compare the whole of the child and mum trees.

[2] Trapping special cases, such as multiplication by zero, and multiplication or division by one, only sped up GP by a few percent.

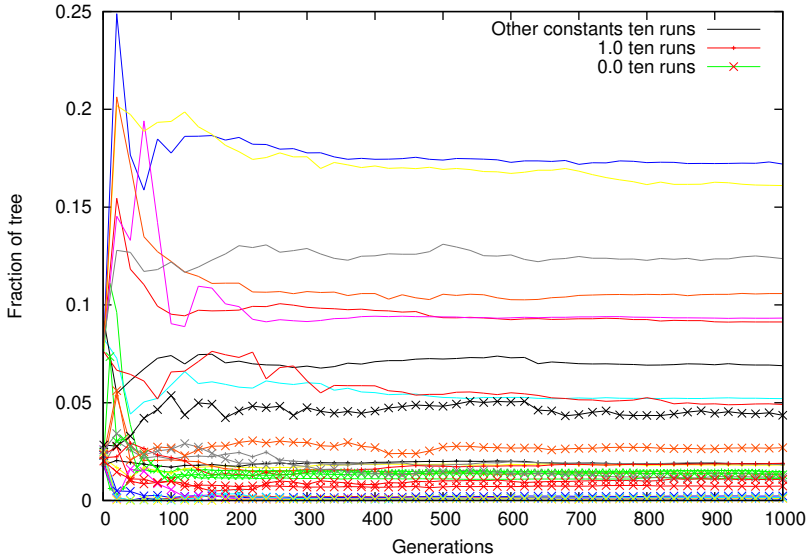

Fig. 3. Evolution of the fraction of functions in GP trees which give the same value (0.0, 1.0 or another constant) on all 48 test cases. Average across population. Tick marks every 20 generations. See also Table 1.

If the subtree replaced by the subtree donated by the dad are identical, then the whole of both trees are identical. In these Sextic polynomial [1] runs, by generation 1000 on average 6% of crossovers swap identical trees.

A second possibility is: what if the crossover subtrees are not genetically identical but yield the same value for all of the fitness cases. For example, we would expect evaluating (ADD X 0.837) and (ADD 0.837 X) to produce identical values. Thus a child created by replacing (ADD X 0.837) with (ADD 0.837 X) or even (SUB (ADD 0.837 X) (SUB X X)) should have have the same fitness as its mum. Thus if we evaluate the child's subtree and re-evaluate the subtree which has just been replaced (from its mum) and for all fitness cases, the mum and child subtrees return the same value, again the child's fitness must be identical to that of the mother. So again, if the evaluation of subtree replaced and the evaluation of the subtree donated by the dad are identical, then the fitness of the whole both child and mum trees are identical. Naturally evaluating two small subtrees is typically much cheaper than evaluating the whole of the child. However crossovers that replace a subtree with a different one which gives the same evaluation are rare (about 0.2% by generation 1000).

What if the evaluation on the test cases of the old and new subtrees are not identical but are similar? We can propagate both vectors of evaluations up the tree towards the root node to the function which calls the modified code. (Figure 4 shows a single test case.) Notice the function itself and its other argument are identical in both mum and child. To minimise cache memory load we can work with just one, typically the child. (Notice as far as this crossover is

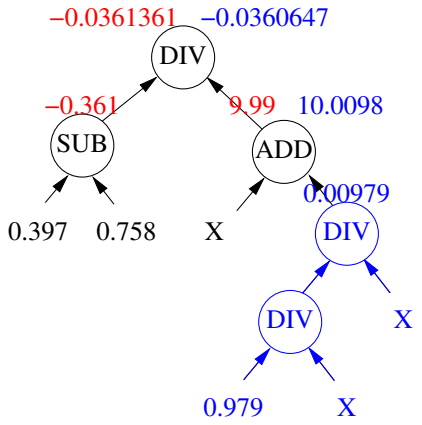

Fig. 4. Incremental evaluation of fragment of child produced by crossover. Inserted subtree (DIV (DIV 0.979 X) X) in blue. Nodes common to both mum and child in black. Red floats (left) are mum node return values. Blue floats (right) are node return values in the child. (In both X=10.0). Notice (SUB 0.397 0.758) is not affected by the crossover and has the same value in parent and child and so is evaluated only once per test case.

concerned we can now discard both parent trees [20].) We evaluate the calling function's other argument once (it must yield the same answers in both mum and child trees). We then evaluate the calling function twice: once for the child and once for the mum. Notice that if for any fitness test case the old and new subtrees gave identical values, then for (at least) those test cases, the evaluation of the calling function must yield the same values. Finally, if the evaluation of the calling function is identical for all test cases, then the evaluation of the whole of the child and mum trees must be identical, hence their fitness values must be the same. So we can stop the evaluation of the child and simply set its fitness to be the same as its mum.

If the mum and child evaluation at the calling function are different for at least one test case, we can repeat the process by proceeding up the tree towards the root node and repeat the evaluation for that calling function. Again we have to evaluate (once per test case) its other argument and the function itself for both child and mum vector of evaluations (created by evaluating the lower function).

In terms of efficiency, the first thing to notice is, if we stop well before the root node, we have not evaluated the vast majority of the child tree. We have evaluated the other sub tree. But we would have had to do that anyway. And we have evaluated the calling function twice per test case. Still a large win, if the evaluation of the child and mum are identical at this point. A possible future work would be to consider if the evaluations for some test cases are different, are the difference important? Is it possible given differences in partial evaluation at this point within the tree to predict if they will have a beneficial or negative impact on fitness. Indeed will the impact on fitness (particularly where approximate

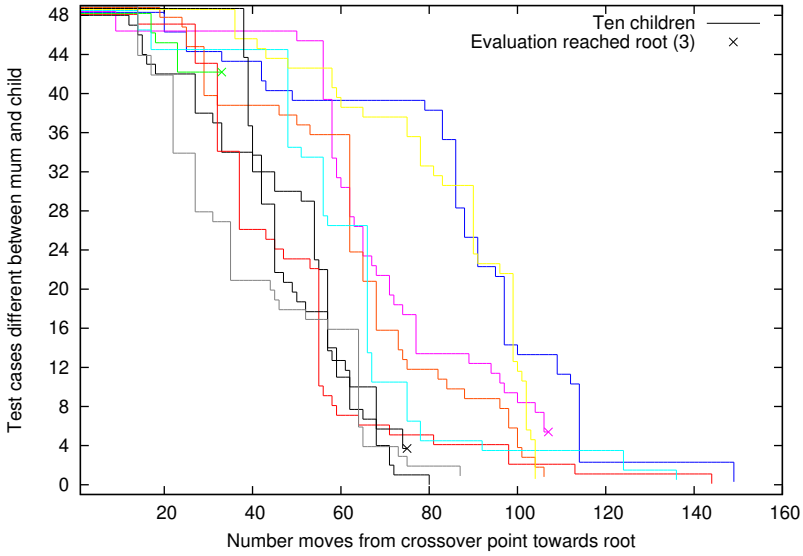

Fig. 5. Incremental evaluation of first ten members of generation 1000. Number of test cases where evaluation in the root donating parent (mum) and its offspring are identical never falls. In three cases (×) evaluation is halted by reaching the root node. Small vertical offset added to separate plots.

answers are good enough [39]) be sufficient to cause a change in breeding pattern for the next, i.e. the grand-child's, generation?

In converged populations there are many cases where trees begat other trees with identical fitness and hence there is hope for this approach to short cut repeated evaluation of what is substantially the same code in the next generation. For example, in Figure 1 on average only 2.6% of trees have fitness different from their mum. Indeed, there are 24 784 generations where everyone in the population has the same fitness as their mum. (After generation 100, Figure 1 smooths the plots by showing the means of 50 generations, so the fraction of changed fitness, lowest dashed black line, is not shown at zero.)

There is a ratchet effect, whereby: if at any point the evaluation of a test case for both mum and child is identical, it will remain identical all the way up to the root node (see Figure 5).

If at no point are all the test cases identical, we will reach the root node. But notice, even in this worst case, the overhead is modest. We will have evaluated the whole of the child, which we would have had to do anyway, plus re-evaluating the (usually small) subtree from the mum, plus evaluating all the functions from the crossover point to the root node. I.e. in the worst case the overhead is less than O(child's height), $\leq$O(size$^{\frac{1}{2}}$) (Section 1). Table 2 shows variation between runs, but even at the start of long runs, saving in terms of opcodes not evaluated can already result in evaluating on average less than half of each tree. As the

Table 2. Mean tree size ("Full") and mean opcodes incrementally ("Inc") evaluated (per test case). "ratio" gives incremental evaluation saving. Ten runs at generation 1000.

	1	2	3	4	5	6	7	8	9	10	mean
Full	70268	14873	34546	220163	138145	36619	35890	26943	19522	35723	63269
Inc	33051	12016	22476	103503	103487	17224	7364	14880	8047	15563	33761
ratio	2.13	1.24	1.54	2.13	1.33	2.13	4.87	1.81	2.43	2.30	2.19

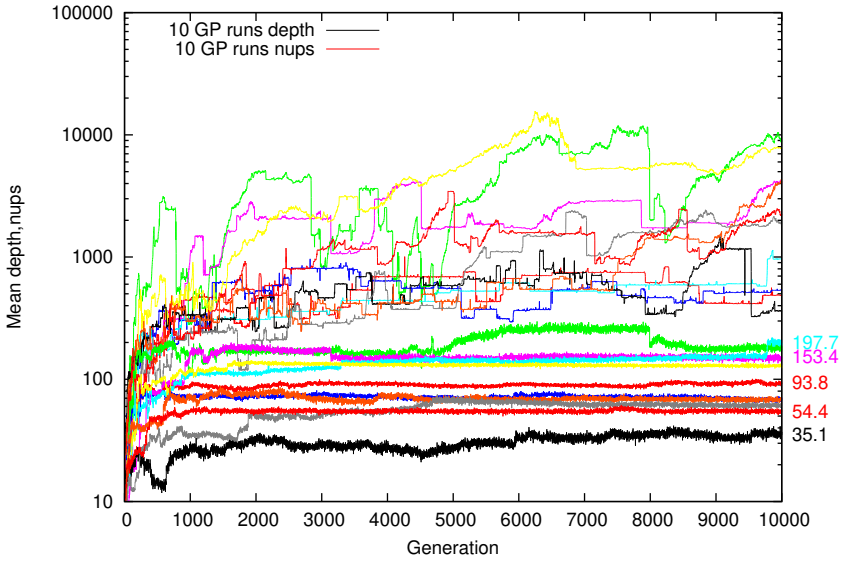

Fig. 6. Evolution of mean depth of trees and number of upward steps (nups) required by incremental evaluation. Notice nups (lower lines) tends to be more stable than tree depth. Ten runs. Last generation summarised in Table 3. Note log vertical scale.

population converges, the saving can grow. For example, in the first run by generation 100 000 on average only $1/70^{\text{th}}$ of each tree is evaluated, Figure 1.

3 Implementing Incremental Evaluation

From a practical point of view it is not hard to implement incremental evaluation. Firstly we need a clean implementation of EVAL, which can be directed to evaluate a subtree, rather than one dedicated to evaluating the whole of a tree. For simplicity, and as we use Intel's Advanced Vector Extensions SIMD instructions (AVX), we run EVAL on all 48 test cases and it returns a vector of 48 floats. (It would be possible to exploit the ratchet effect, Section 2, by keeping track of which fitness cases mum and child evaluations are different and ensuring EVAL does not executes those that are the same.)

For IncFit we need the child tree, the old crossover fragment removed from the mum, and the location of the crossover point in the child. We also need an

Table 3. Mean tree sizes, mean depths and mean number of upward steps, in ten GP runs at generation 10 000. ± indicates population standard deviation. The last column gives means of the ten runs. Data from Figure 6.

run:	1	2	3	4	5
size/1000	349±4	8655±5813	276±4	5066±1095	997±4
depth	486±2	8801±2230	534±8	3985±839	964±3
nups	94±42	172±127	68±32	153±98	198±126

run:	6	7	8	9	10	mean
size/1000	9171±1345	98±2	4653±1075	5232±1548	1567±173	3606
depth	7844±666	359±2	4133±782	1826±178	2228±512	3116
nups	130±51	35±34	69±40	60±33	54±29	103

efficient way to navigate up the tree to the function calling a given subtree and to find its other subtree.

We start by calling EVAL for both the crossover fragment in the child tree and for the crossover fragment in the mum tree. Each call to EVAL returns an array of 48 floats which we bitwise compare with memcmp. If they are identical, we stop and set the child's fitness to that of the mum tree. If not, we go up one level in the child tree. (For the rest of the new tree's evaluation, it is effectively identical to its mum tree.) We locate the upper function's other argument and call EVAL for it. (I.e., we EVAL the other subtree.) We now have three vectors each of 48 floats.

The next thing to do is to evaluate the function (ADD, SUB, MUL or DIV), giving it two vectors of floats (*in order*). Remember that the other subtree may be either the function's first or second argument. For this you perhaps want a specialised version of EVAL. evalop, rather than having to recursively evaluate its two arguments, it has them passed to it directly. Our evalop takes two 48 float vectors and returns a 48 float vector of results. We call it twice, once with the other subtree's vector and that from the mum and a second time again with the other subtree's vector and this time with the child's evaluation vector.

We compare evalop's two output vectors (again using memcmp). If they are the same we stop and use the mum's fitness, otherwise we proceed up the tree one more level and repeat. If we reach the root node, we use the child's vector of 48 floats to calculate the fitness of the child.

The GP parameters are given in Table 4. We run up to ten experiments with different pseudo random number seeds.

4 Discussion: Population Convergence

4.1 Why Does Incremental Evaluation Work?

We have a chicken and egg situation. For this (exact) version of incremental evaluation to work, we need the GP population to show significant convergence, with many children resembling their parents (i.e. have identical fitness). For convergence, we may need long runs, which implies the need for fast fitness

Table 4. Evolution of Sextic polynomial symbolic regression binary trees

Terminal set:	X, 250 constants between -0.995 and 0.997
Function set:	MUL ADD DIV SUB
Fitness cases:	48 fixed input -0.97789 to 0.979541 (randomly selected from -1.0 to +1.0). Target $y = xx(x-1)(x-1)(x+1)(x+1)$
Selection:	Tournament size 7 with fitness $= \frac{1}{48}\sum_{i=1}^{48} \lvert GP(x_i) - y_i \rvert$
Population:	500. Panmictic, non-elitist, generational.
Parameters:	Initial population ramped half and half [1] depth between 2 and 6. 100% unbiased subtree crossover. At least 1000 generations

DIV is protected division `(y!=0)? x/y : 1.0f`

evaluation, which incremental evaluation may provide. With many children with unchanged fitness, there is hope that chasing up the tree from the crossover point (i.e. the source of disruption) towards the root node, the difference between child and parent will dissipate.

This relies on the function set and test set being dissipative, i.e. losing information. But if the functions are perfect, i.e. never lose any information (and so are reversible), then convergence and indeed evolution is impossible [40]. However, traditional GP systems do evolve, their function sets are not reversible, they do lose information. Small GP populations, if allowed time and space, can show elements of convergence, in which case this form of incremental evaluation may flourish. Indeed by applying information theory to GP function sets, we may be able to design more evolvable GP systems.

By generation 1000 the populations have bloated and on average incremental evaluation has to process about 100 nested function calls, Table 3. Figure 8 and Table 5 show that the change in the difference between mum and her offspring as we evaluate each opcode during incremental fitness evaluation. The data are grouped by run and by function. There is a clear distinction between the linear functions, ADD and SUB, and the non-linear functions, MUL and DIV.

Most linear functions do not noticeably change the difference between the parent evaluation and that of the child. This is expected. E.g. if on each of the 48 test cases, the difference between the re-evaluation of the root donating parent (the mum) and that of its offspring is 1.0, we would expect after ADD, the difference would remain 1.0 no matter what ADD's other argument. However ADD is a floating point operator and so is potentially subject rounding errors. Suppose on one test case, ADD's other argument is 10^{23}, it may be that ADD's output, for both mum and child is 10^{23} and therefore on that test case their difference is now zero. Notice that ratchet effect does not apply to differences, only to the special case of values being identical (Section 2), and while differences tend to decrease, monotonic decrease is not guaranteed.

With non-linear operations we expect them to change the difference between mum and offspring evaluations. E.g. if MUL's other argument was 2.0 for all test cases, we would expect the difference between mum eval and child eval to double for every test cases. Table 5 largely confirms this and shows that only about 1% of MUL or DIV operations do not change the difference. Indeed MUL

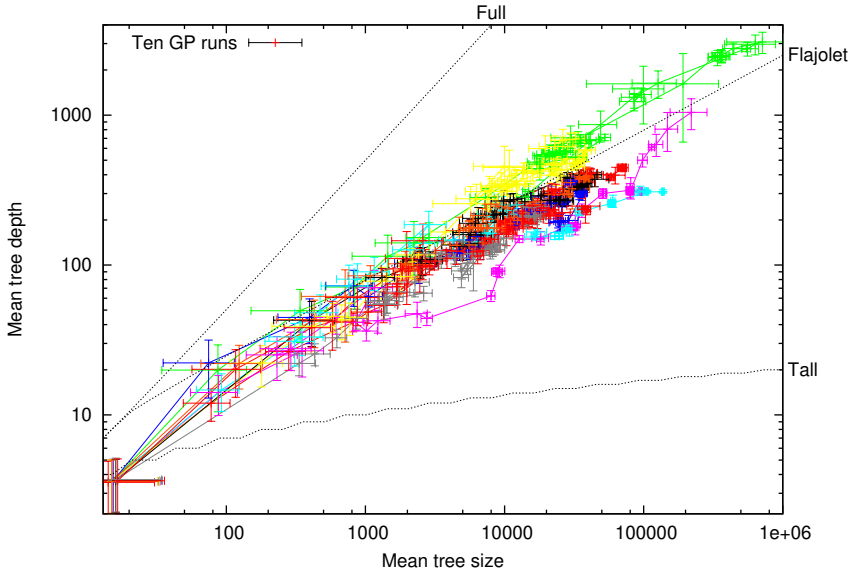

Fig. 7. Evolution of mean program size and depth in ten runs of Sextic polynomial up to generation 1000. Cross hairs show population standard deviations every 20 generations. Dotted lines show tree limits. Flajolet line is height of random trees $\sqrt{2\pi(\text{size})}$ [27]. Note log scales.

and, in particular, DIV can magnify the difference enormously (dotted purple line in Figure 8). However there are more operations which reduce the difference than increase it. Rounding error is still active and causing information loss and although differences may rise or fall, once they reach zero on a particular test case, the difference on that test case must remain zero (Section 2). Typically over about one hundred evolved function calls the difference on all 48 test cases is zero and incremental fitness evaluation can stop.

There are of course considerable differences between individual crossovers but averaging at generation 1000 over the whole population and ten runs, shows on average each ADD reduces the number of test cases which are different by 13%, and the other functions by similar amounts: SUB 12%, MUL 10% and DIV 19%. If we grossly simplify by assuming the test cases are independent and there is a fixed chance of an operation clearing the difference for a given test case. The chance of test case difference being non-zero will fall geometrically (p^{nups}), with number of moves up the tree, nups. The chance of all test cases having non-zero difference is $(p^{\text{nups}})^{48}$, which is also geometric. Using the mean of the geometric distribution (here $1/p^{48}$) and the observed mean number of steps required (103.433800), and taking logs of both sides gives $-\log(p^{48}) = \log(103.433800)$ rearranging $\log(p) = -\log(103.433800)/48$ = -0.0966444. This suggests each upward move has about a 9% chance of synchronising the evaluation of a test case. I.e. each function destroys about three bits of information

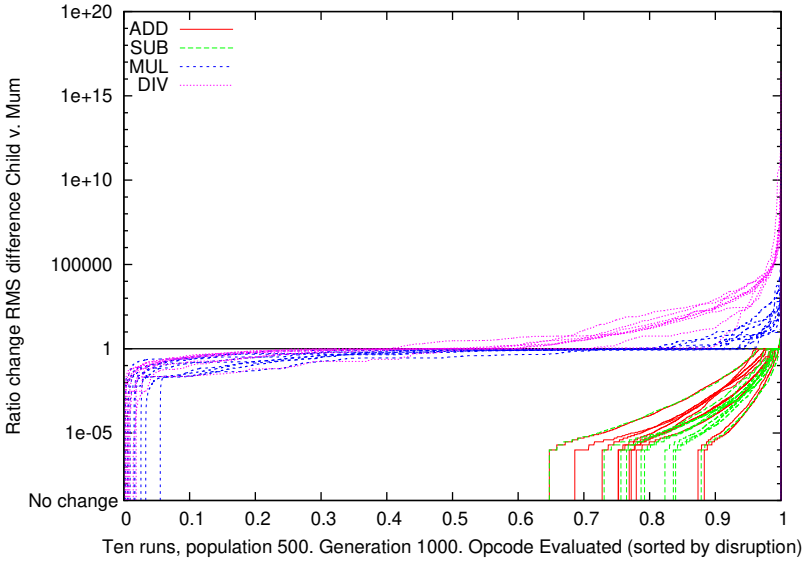

Fig. 8. Ratio of difference between mum and child before and after each function at generation 1000. Most linear functions (i.e. ADD and SUB) do not change difference. On average 86% MUL and 56% DIV decrease difference. X-axis normalised to allow easy comparison between ten runs. Note log scale.

per test case. Rather more than the minimum 0.5 bits expected of well behaved floating point rounding, but similar to the last row (nups) of Table 3.

As Figure 9 shows incremental fitness evaluation does better as the population converges and trees become larger.

4.2 Implications for Software Engineering

By software engineering standards our pure functions, even though large by GP standards, are simple. Nevertheless it is interesting that even such a pure functional program loses information and our analysis allows us to track dissipation of disruption from its cause (in the case of GP, the crossover point). Like human written software the disruption may or may not lead to a visible external affect. (In the case of GP, to be visible, the disruption must reach the root node.)

Although information loss is a general effect, in Software Engineering it is known mostly by its specific effects. In mutation testing, inserted bugs that cause no visible impact are known as equivalent mutants [41,42,43]. One off run time bit flips which fail to impact the program's output are known as correctness attraction [44], indeed Danglot et al. say programs are Antifragile. Coincidental correctness is another term used to describe when a program produces a correct answer despite an error in its source code, even though the error has been executed [45]. Bugs which have no visible impact (so far) are known as latent bugs [46]. The PIE (propagation, infection, and execution) view [47] considers

Table 5. Percentages of functions in GP trees which leave unchanged, reduce or increase difference between mum and offspring in ten GP runs at generation 1000. Last column gives means of the ten runs. Data from Figure 8.

	run	1	2	3	4	5	6	7	8	9	10	mean
No change	ADD	75.21	88.31	75.25	77.19	77.96	72.76	68.52	87.34	76.88	64.72	76.41%
	SUB	82.32	87.84	83.94	76.48	73.07	78.69	75.62	83.60	79.20	64.71	78.55%
	MUL	0.52	1.68	0.18	3.33	1.54	0.65	5.52	0.19	0.93	2.59	1.71%
	DIV	0.63	1.77	0.14	na	na	0.33	0.84	0.29	0.58	1.40	0.75%
Reduce	ADD	24.34	11.60	24.52	22.62	21.77	26.81	31.39	12.47	22.62	34.60	23.27%
	SUB	17.50	12.03	15.80	23.33	26.84	21.08	23.96	16.21	20.54	34.76	21.21%
	MUL	94.84	83.80	91.22	73.40	78.25	92.30	85.16	94.21	82.13	87.84	86.32%
	DIV	39.43	69.33	47.49	na	na	58.96	50.99	54.94	70.05	58.03	56.15%
Increase	ADD	0.45	0.09	0.23	0.19	0.26	0.43	0.09	0.19	0.50	0.67	0.31%
	SUB	0.18	0.13	0.26	0.19	0.09	0.23	0.42	0.19	0.26	0.53	0.25%
	MUL	4.64	14.52	8.60	23.27	20.20	7.05	9.31	5.59	16.94	9.57	11.97%
	DIV	59.94	28.90	52.37	na	na	40.72	48.18	44.77	29.37	40.57	43.10%

the impact of bugs and if it propagates to the program's outputs. Failed error propagation [48] acknowledges the PIE frameworks and shows that the impact of bugs can be hidden by entropy loss and explained in terms of information theory. These can be summarised as the robustness of software [49].

5 Conclusion: Information Loss is Essential to Converge

Incremental fitness evaluation can be easily implemented and in even the worst case imposes little overhead. With very large trees, it can speed up GP by an order of magnitude. Using 48 cores of a 3.0GHz server gave the *equivalent* of 749 billion GPop/s, exceeding the best direct interpreters [50, Tab. 3] [51] [34].

We had assumed that the considerable bloat seen here was due to simple introns, such as multiplication or division by zero, and so runtime could be reduced by exploiting obvious fitness evaluation short cuts. However this produced only a 1% saving. Instead something much more interesting is happening.

Information flows from the leafs of GP trees via arithmetic functions to the root node. These functions are not reversible and inevitably loose information. This information loss gives rise to semantic convergence. That is, if we regard GP children as being the same as their parent plus a syntactic change (or error) made by crossover or mutation and trace fitness evaluation from the error to the root node, we see the values flowing in the parent and in the child to be increasingly similar. If the trees are large these information flows can becomes identical. If the information flow leaving the child and parent are identical then their fitness is also identical. Note GP has evolved a smooth fitness landscape.

Many GP systems do not have side effects and so their trees can be evaluated in any order. Our incremental approach evaluates trees by following the information flow. It can stop early when the data flow in the child is the same as that in its parent.

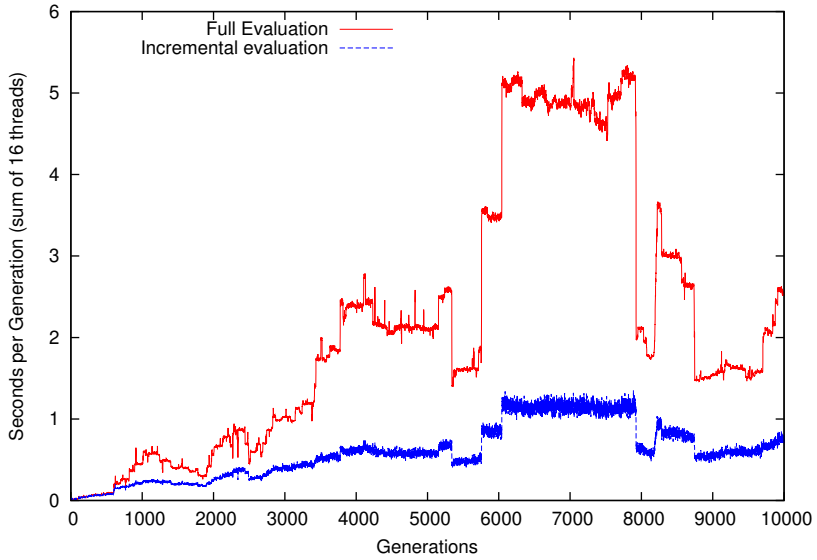

Fig. 9. Time per generation for top down recursive and bottom up incremental evaluation. 16 core 3.80GHz i7-9800X, g++ 9.3.1. Performance equivalent of 53.1 billion GP operation per second (full, top down) and 179 10^9 GPop/s (incremental, bottom up).

It is possible to measure the disruption to the information flow caused by the injected code. From that it may be possible to predict in advance, e.g. given the size and nature of the parent tree and the location of the injected error, how much disruption will reach the child's root node and so estimate if the injected code will change its fitness and if so by how much. Turning this about: from such an information based model, it may be possible to choose where to place new code in highly fit trees to avoid simply reproducing exactly the same fitness and perhaps even avoiding changes which will cause fitness to fall enormously.

Acknowledgements

Funded by EPSRC grant EP/P005888/1.
Code `http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/gp-code/GPinc.tar.gz`

References

1. Koza, J.R.: Genetic Programming: On the Programming of Computers by Natural Selection (1992)
2. Langdon, W.B., Poli, R.: Why ants are hard. In: Koza, J.R., et al. (eds.) GP. pp. 193–201 (1998)
3. Teller, A.: The internal reinforcement of evolving algorithms. In: Spector, L., et al. (eds.) AiGP 3, pp. 325–354 (1999)

4. Langdon, W.B.: Genetic Programming and Data Structures (1998)
5. White, D.R., et al.: Evolutionary improvement of programs. IEEE TEVC 15(4), 515–538 (2011)
6. Langdon, W.B., Harman, M.: Optimising existing software with genetic programming. IEEE TEVC 19(1), 118–135 (2015)
7. Petke, J., et al.: Using genetic improvement and code transplants to specialise a C++ program to a problem class. In: Nicolau, M., et al. (eds.) EuroGP. LNCS, vol. 8599, pp. 137–149 (2014)
8. Petke, J.: Constraints: The future of combinatorial interaction testing. In: SBST. pp. 17–18 (2015)
9. Petke, J., et al.: Specialising software for different downstream applications using genetic improvement and code transplantation. TSE 44(6), 574–594 (2018)
10. Petke, J., et al.: Genetic improvement of software: a comprehensive survey. IEEE TEVC 22(3), 415–432 (2018)
11. Takagi, H.: Interactive evolutionary computation: Fusion of the capabilities of EC optimization and human evaluation. Proc. IEEE 89(9), 1275–1296 (2001)
12. Backer, G.: Learning with missing data using genetic programming. In: The 1st Online Workshop on Soft Computing (WSC1). Nagoya University, Japan (1996)
13. Poli, R., et al.: A field guide to genetic programming. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk` (2008)
14. Langdon, W.B., Buxton, B.F.: Evolving receiver operating characteristics for data fusion. In: Miller, J.F., et al. (eds.) EuroGP. LNCS, vol. 2038, pp. 87–96 (2001)
15. Gathercole, C., Ross, P.: Dynamic training subset selection for supervised learning in genetic programming. In: Davidor, Y., et al. (eds.) PPSN. LNCS, vol. 866, pp. 312–321 (1994)
16. Spector, L.: Assessment of problem modality by differential performance of lexicase selection in genetic programming: A preliminary report. In: McClymont, K., Keedwell, E. (eds.) GECCO Comp. pp. 401–408 (2012)
17. Teller, A., Andre, D.: Automatically choosing the number of fitness cases: The rational allocation of trials. In: Koza, J.R., et al. (eds.) GP. pp. 321–328 (1997)
18. Ni, J., et al.: The use of an analytic quotient operator in genetic programming. IEEE TEVC 17(1), 146–152 (2013)
19. Koza, J.R., et al.: Genetic Programming III: Darwinian Invention and Problem Solving (1999)
20. Langdon, W.B.: Multi-threaded memory efficient crossover in c++ for generational genetic programming. SIGEVOlution 13(3), 2–4 (2020)
21. Singleton, A.: Genetic programming with C++. BYTE pp. 171–176 (1994)
22. Langdon, W.B.: Parallel GPQUICK. In: Doerr, C. (ed.) GECCO Comp. pp. 63–64 (2019)
23. Keith, M.J., Martin, M.C.: Genetic programming in C++: Implementation issues. In: Kinnear, Jr., K.E. (ed.) AiGP, pp. 285–310 (1994)
24. Handley, S.: On the use of a directed acyclic graph to represent a population of computer programs. In: WCCI. pp. 154–159 (1994)
25. McPhee, N.F., et al.: Sutherland: An extensible object-oriented software framework for evolutionary computation. In: Koza, J.R., et al. (eds.) GP. p. 241 (1998)
26. Ehrenburg, H.: Improved directed acyclic graph evaluation and the combine operator in genetic programming. In: Koza, J.R., et al. (eds.) GP. pp. 285–291 (1996)
27. Sedgewick, R., Flajolet, P.: An Introduction to the Analysis of Algorithms (1996)
28. Langdon, W.B.: Fast generation of big random binary trees. Tech. Rep. RN/20/01, Computer Science, University College, London (2020)

29. Langdon, W.B., Banzhaf, W.: Continuous long-term evolution of genetic programming. In: Fuechslin, R. (ed.) ALIFE. pp. 388–395 (2019)
30. Langdon, W.B., Banzhaf, W.: A SIMD interpreter for genetic programming on GPU graphics cards. In: O'Neill, M., et al. (eds.) EuroGP. LNCS, vol. 4971, pp. 73–85 (2008)
31. Langdon, W.B., Harrison, A.P.: GP on SPMD parallel graphics hardware for mega bioinformatics data mining. Soft Computing 12(12), 1169–1183 (2008)
32. Poli, R., Langdon, W.B.: Sub-machine-code genetic programming. In: Spector, L., et al. (eds.) AiGP 3, pp. 301–323 (1999)
33. Langdon, W.B.: Long-term evolution of genetic programming populations. In: GECCO. pp. 235–236 (2017)
34. Langdon, W.B.: Genetic improvement of genetic programming. In: Brownlee, A.S., et al. (eds.) GI @ CEC 2020 Special Session (2020)
35. Langdon, W.B., et al.: The evolution of size and shape. In: Spector, L., et al. (eds.) AiGP 3, pp. 163–190 (1999)
36. Altenberg, L.: The evolution of evolvability in genetic programming. In: Kinnear, Jr., K.E. (ed.) AiGP (1994)
37. Tackett, W.A.: Recombination, Selection, and the Genetic Construction of Computer Programs. Ph.D. thesis (1994)
38. Langdon, W.B., Poli, R.: Fitness causes bloat. In: Chawdhry, P.K., et al. (eds.) Soft Computing in Engineering Design and Manufacturing. pp. 13–22 (1997)
39. Mrazek, V., et al.: Evolutionary approximation of software for embedded systems: Median function. In: Langdon, W.B., et al. (eds.) GI. pp. 795–801 (2015)
40. Langdon, W.B.: The distribution of reversible functions is Normal. In: Riolo, R.L., Worzel, B. (eds.) GPTP, pp. 173–187 (2003)
41. Yao, X., et al.: A study of equivalent and stubborn mutation operators using human analysis of equivalence. In: Briand, L., et al. (eds.) ICSE. pp. 919–930 (2014)
42. Jia, Y., et al.: Learning combinatorial interaction test generation strategies using hyperheuristic search. In: Bertolino, A., et al. (eds.) ICSE. pp. 540–550 (2015)
43. Langdon, W.B., et al.: Efficient multi-objective higher order mutation testing with genetic programming. Journal of Systems and Software 83(12), 2416–2430 (2010)
44. Danglot, B., Preux, P., Baudry, B., Monperrus, M.: Correctness attraction: A study of stability of software behavior under runtime perturbation. Empr. Soft. Eng. 23(4), 2086–2119 (2018)
45. Abou Assi, R., et al.: Coincidental correctness in the Defects4J benchmark. Soft. TVR 29(3), e1696 (2019)
46. Timperley, C.S., et al.: Crashing simulated planes is cheap: Can simulation detect robotics bugs early? In: ICST. pp. 331–342 (2018)
47. Voas, J.M., Miller, K.W.: Software testability: The new verification. IEEE Software 12(3), 17–28 (May 1995)
48. Clark, D., et al.: Normalised squeeziness and failed error propagation. Info. Proc. Lets 149, 6–9 (2019)
49. Langdon, W.B., Petke, J.: Software is not fragile. In: Parrend, P., et al. (eds.) CS-DC. pp. 203–211 (2015)
50. Langdon, W.B.: Large scale bioinformatics data mining with parallel genetic programming on graphics processing units. In: Tsutsui, S., Collet, P. (eds.) Massively Parallel Evolutionary Computation on GPGPUs, pp. 311–347 (2013)
51. de Melo, V.V., et al.: A MIMD interpreter for genetic programming. In: Castillo, P.A., et al. (eds.) EvoApps. LNCS, vol. 12104, pp. 645–658 (2020)