Genetic Programming and Evolvable Machines (2022) 23:71–104
https://doi.org/10.1007/s10710-021-09405-9

Genetic Programming Convergence

W. B. Langdon

7 June 2021 *Revision* : 1.147

Abstract We study both genotypic and phenotypic convergence in GP floating point continuous domain symbolic regression over thousands of generations. Subtree fitness variation across the population is measured and shown in many cases to fall. In an expanding region about the root node, both genetic opcodes and function evaluation values are identical or nearly identical. Bottom up (leaf to root) analysis shows both syntactic and semantic (including entropy) similarity expand from the outermost node. Despite large regions of zero variation, fitness continues to evolve and near zero crossover disruption suggests improved GP systems within existing memory use.

Keywords: evolutionary computation, stochastic search, diversity, bottom up incremental evaluation, PIE, propagation, infection, and execution, SIMD parallel processing, AVX vector instructions

1 Introduction

1.1 Summary

Since this is a long paper, perhaps unusually, we shall start with a brief summary and sign posting of what is to come and the paper's goals, novelty, impact and contributions.

Much Genetic Programming work is aimed at applications where there is a need for a quick solution and so GP runs tend to be short, e.g. no more than fifty generations. But the goal of GP should also be to solve problems which cannot be solved by other methods. Recently Rich Lenski has confounded the Biological establishment and overturned conventional wisdom by showing that natural evolution can continue to produce fitter organisms even after tens of thousands of generations (see Section 1.5). Perhaps a way to open up GP to

more adventurous applications, will require that the GP population evolves for far longer?

We have made a start with long term evolution experiments in GP. We have run GP populations far longer than previously attempted. These have shown, even in fixed environments, GP can continue to find improvements. Our goal here is to understand in as much detail as possible what is going on in these highly evolved populations. In the process we have devised tools (which are available via my home page) which substantially speed up (and indeed make feasible) long experimental runs and indeed may be useful in large GP experiments even in short runs. Results indicate that most of the GP fitness landscape is far smoother than commonly assumed, with crossover becoming less phenotypically disruptive as tree grow larger and initial models, in Section 9.6, suggesting increasing the rigour of the fitness function will only slowly increase crossover's effect.

In Section 2 we summarise the GP system recently used and the results obtained. Whilst the following sections start the detailed analysis of these evolved populations. Section 3 investigates genetic convergence. Section 4 shows that phenotypic convergence lags behind genetic convergence of the trees. Section 5 shows although operations like multiplication or division by zero can render large parts of trees ineffective, in the continuous domain such obviously ineffective code can be a small ($\approx$0.5%) or a large (91%) part of highly evolved programs. Thus explaining why automatic intron removal may not always improve GP execution time. Section 6 shows in converged populations many subtrees have identical phenotypes. In Section 7, in contrast to ordinary human written computer programs, by studying information flow within evolved trees we find on average entropy *rises* monotonically from the inputs (the leafs) towards the the output (the tree's root node) and quickly reaches a maximum, $\log_2$ |test suite size|. This means large parts of evolved programs have identical entropy. In contrast, Section 8 shows typically the phenotypic disruption of crossover has a limited effect, which tends to be damped in the region above the crossover point towards the root node. This opens the way to the implementation of new efficient GP interpreters for large evolving programs. Finally, Section 9 describes a few of the limitations of our approach before we conclude in Section 10.

1.2 Background: Convergence in Population Based Search

In all optimisation and machine learning there is an exploration-exploitation balance to be struck between seeking new solutions (exploring) and exploiting the best results founds so far by searching in their neighbourhood. In evolutionary computing population convergence gives a measure of how the balance has been struck so far. A highly converged population suggests, search has been mostly locally exploiting the neighbourhood of the best seen so far. Whilst a diverse population suggests search is more explorative. Studying population

0000000000
1000000000
0100000000
0010000000
0001000000

Fig. 1 Example of a converged population of five bit strings. In the 4 leftmost positions at most one member of the population is different from the rest, whilst in the remainder they are all the same. Thus although each member of the population is unique, yet they are similar. Studying bit positions makes this clear, and having a simple fixed representation makes it easy to do.

diversity may give insights into how well a search technique is performing or suggest ways to improve it.

In the case of fixed representations, such as fixed length bit strings [1,2], convergence is often defined by summing the lack of variation in individual bit positions in the evolved population. This has the advantage of capturing convergence even when every bit string in the population is different but they each have many bit positions in common (see Figure 1).

1.3 Background: Convergence of Genetic Programming Populations

Some evolutionary computing techniques, in particular genetic programming (GP) [3,4], allow the representation [5] to evolve. Although there are many types of GP [6,4,7,8,9,10,11], and more recently genetic improvement has been applied to fix bugs [12] in human written code or otherwise increase its performance [13][14][15][16] [17][18][19][20][21], we will concentrate on Koza's evolution of lisp s-expressions in the form of trees [3]. We will study the fixed arity case where the tree's internal nodes (functions) have exactly two arguments. In particular, we will use the traditional four GP mathematical functions addition, subtraction, multiplication and protected division [3][1]. Again, to simplify, we follow Koza's [3] definition of subtree crossover, but select the crossover points uniformly at random[2]. From these four functions it is possible *in principle* to construct any polynomial and any continuous function can be approximated by a polynomial.

Perhaps due to the difficulty of defining convergence of trees, convergence of bit string GAs has been more studied than that of GP populations. Koza [3] defined a population *variety* measure as being the fraction of unique individuals in the population. He showed in evolved GP populations variety remains near 100% [3]. As with bit string GAs (Figure 1), we can consider convergence of components of trees [22,23]. Koza, O'Reilly [24], and in particular Poli [25,26, 27,28,29], used mathematical models to analyse the short term evolution of program fragments in GP populations. They devised various ways to partition

[1] We define protected division so that division by zero yields a defined result, i.e. 1.0

[2] In [3] Koza uses a 90% bias in favour of internal nodes to reduce the fraction of crossovers which simply move leafs within the trees.

trees into *schema* [30,31] and created increasingly accurate inequalities and then equations to predict their frequency in the next generation [3]. Daida et al. [38,39] also considered the geometry of the binary tree search space on GP as well as various ways to visualise it [40]. Gustafson et al. [41] consider GP population diversity and entropy. (Note Gustafson et al. consider entropy of the population, rather than, as in Section 7, within GP trees.) A number of successful extensions to GP have been proposed which are inspired at least in part by theoretical considerations such as the Schema theorems. For example Poli and Page's smooth uniform crossover (GP-UX) [42], Hansen's homologous crossover [43], Moraglio and Poli's geometric crossover [44] and Pawlak's local and approximately geometric semantic crossovers (LGX and KLX) [45]. These take note of the special significance of the root node in GP trees.

1.4 Background: Crossover

In linear representations, crossover can be assumed to be beneficial by allowing search to combine good components [1]. In diverse populations, crossover can thus be viewed as a long range search operation, which may jump from a good region of the search space over bad regions to a distant good point [46]. The effectiveness of crossover in GP remains controversial [47] and this has prompted studies looking into the effectiveness of Koza's subtree swapping crossover or the value of schema in GP populations, for example [48,49,50, 51,52,53]. Finally some GP benchmarks have been subjected to more or less detailed mathematical run time and convergence analysis [54,55,56,57,58].

1.5 Background: Inspiration from Biology and Earlier GP

Although Biologists are Darwinists [59] at heart, it is often assumed that evolution was something that took a long time and occurred in the past. Yet studies of fish in the African great lakes showed that new species have evolved in the last few thousand years (rather than taking millions of years) [60], and Rich Lenski's experiments with E. coli [61] show even in the most stable of environments bacteria can still improve their metabolisms after tens of thousands of generations [62]. McPhee and Poli [63, Fig. 1a v. 1b] showed, even in something as artificial, simple and mechanical as linear GP, extended evolution can throw up surprises. Recently we have been studying small genetic programming populations run far longer than usual: for thousands, even hundreds of thousands, of generations.

[3] Schema theory has proved quite popular with GP authors [32,33,34,35,36,37].

1.6 Background: Assistance from Parallel Hardware

In [64,65] we used Poli's sub-machine code GP [42,66] to evaluate 64 fitness cases in parallel and so speed up the evolution of populations of 500 binary trees allowing for the first time GP evolution for 100 000 generations[4].

The advent of AVX parallel vector instructions allows parallel evaluation of 16 floating point fitness cases. Again parallel hardware allows us to evolve small genetic programming populations for many thousands of generations. We have use Koza's sextic polynomial [3] as a convenient continuous symbolic regression benchmark [67,68][5] [6]. The sextic polynomial has often used been used as an example GP benchmark for symbolic regression, which remains one of the most popular applications of GP. It uses four functions ($+$, $-$, $\times$ and "protected" division) which are together sufficient in principle to express any polynomial and thus in theory are able to approximate any continuous function.

2 Sextic Polynomial, GPavx

Our GP experiments greatly extend [68] but this section primarily gives the background for the extensive analysis of the first 10 000 generations for a total of five runs in Section 3 onwards. (To be consistent, where data from five runs are plotted together, we have used the same colour/line styles: Figures 5, 6, 10, 11, 13, 16, 19, 21, 22.)

Our experiments use Koza's sextic polynomial (see Figure 2). The 48 fitness test cases are shown with crosses ($\times$) in Figure 2) and Table 1 gives the GP parameters. These are the same as our conference paper [68] but we run for far longer and subject the populations to far more detail analysis. Evolution of tree sizes over 35 days is shown in Figure 3 [7].

In Figure 4 we plot the corresponding convergence of (phenotypic) fitness (averaged over 100 generations). Showing both the fall in the number of children whose fitness is not identical to the best in the population (red solid line) and also the rise in the number of times where everyone in the population has the same fitness (blue dashed line). In 884 of the last 1000 generations (88.4%) all 500 trees have the same fitness.

[4] See http://www.cs.ucl.ac.uk/staff/W.Langdon/gggp/#Langdon:2017:GECCO for animations of the evolution of convergence in binary 6-multiplexor populations. (Also YouTube video: https://youtu.be/gwCwvwJcWbQ.) C++ code available from http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/gp-code/GPbmux6.tar.gz.

[5] See http://www.cs.ucl.ac.uk/staff/W.Langdon/seminars/aigp3/ for an animation of [67, Figure 8.5].

[6] Koza [3] uses 50 fitness cases. However, to run conveniently on AVX-512 hardware we replaced 50 by the closest multiple of 16, i.e. $3 \times 16 = 48$ test cases. Therefore we actually bank together sets of three AVX-512 instructions to evaluate 48 fitness cases together [69, 70,71].

[7] Run aborted due to external power failure. We used GPavx, which on publication [69,70], was the world's fastest general GP system `http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/gp-code/GPavx.tar.gz`, on a 3.00GHz Intel Xeon Gold 6136 server with 48 cores and 3 TBytes. See also RN/20/01 [72]

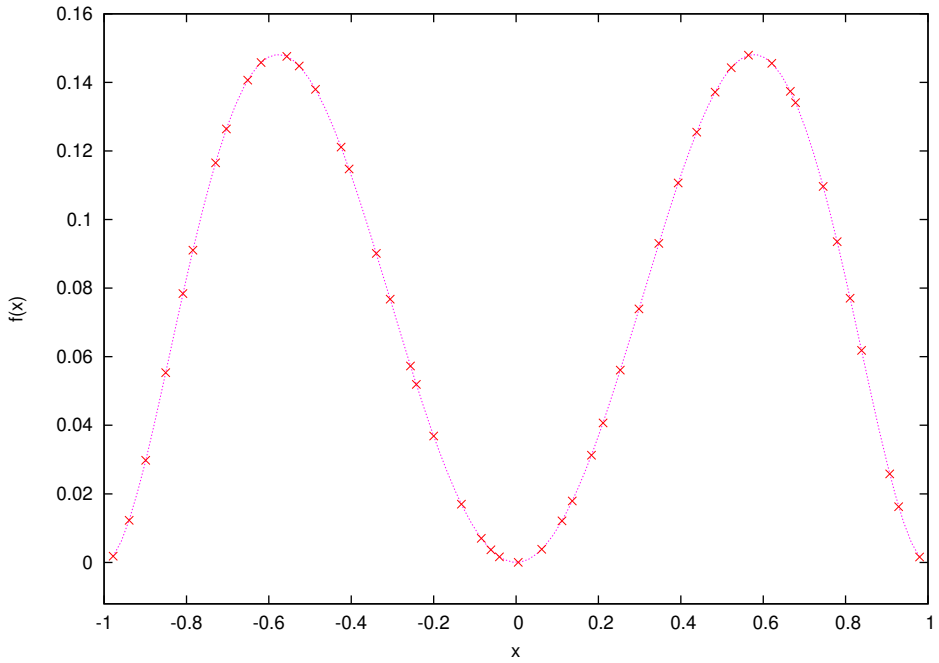

Fig. 2 48 test cases for Sextic Polynomial Benchmark.

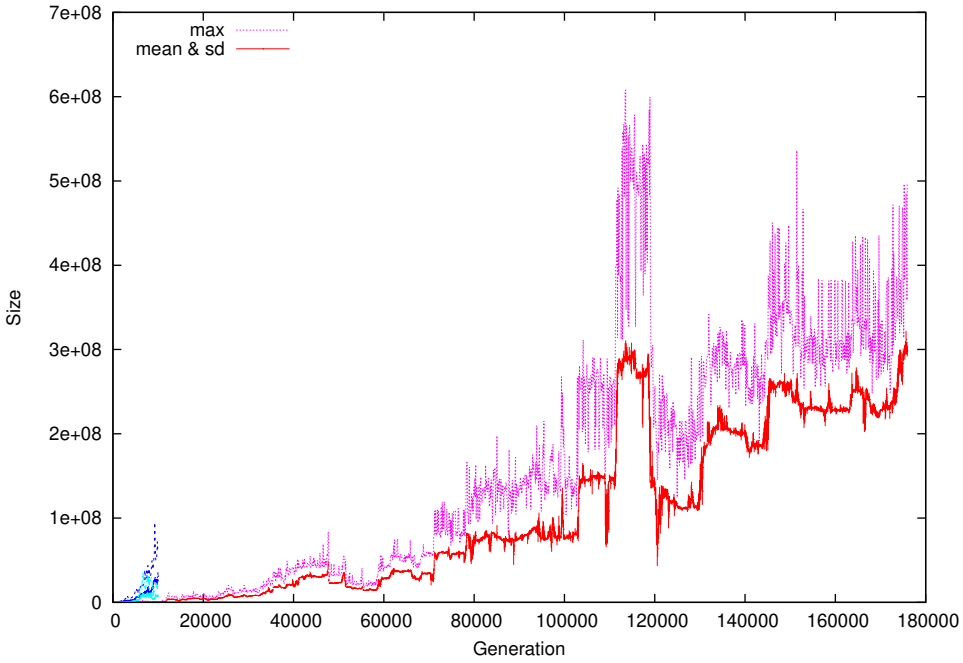

Fig. 3 Evolution 175 973 generations in 5 weeks. Other runs to generation 10 000. (Data averaged over 100 generations.)

Table 1 Long Term Evolution Experiment with Sextic Polynomial

Terminal set:	X, 250 constants -0.995 to 0.997
Function set:	MUL ADD DIV SUB (all binary)
Fitness cases:	48 fixed input -0.97789 to 0.979541 (randomly selected from -1.0 to +1.0 input). $y = xx(x-1)(x-1)(x+1)(x+1)$
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 (500) ramped half and half [3] depth between 2 and 6. 100% unbiased subtree crossover (no mutation). Five runs.

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

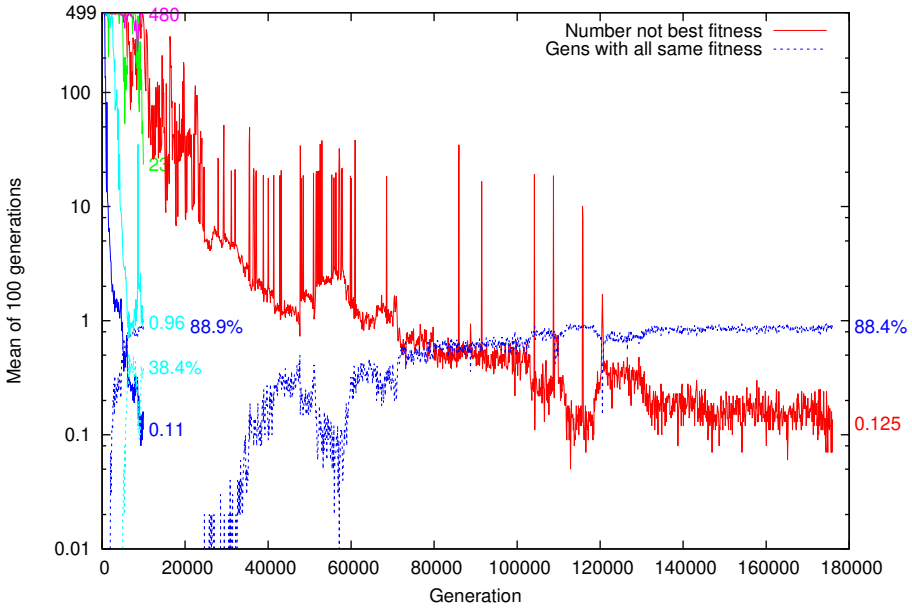

Fig. 4 Evolution of population fitness convergence. After generation 120 000 in most generations everyone has exactly the same fitness, rising to 88.4% on average. Other runs only to generation 10 000. (Note log scale.)

Figure 5 plots the evolution of the best fitness in the population (note log scales). Figure 5 shows during the longest run GP found 3588 progressive improvements. Whilst [68] suggested that crossover might continue to find improvements, the long flat region at the right of Figure 5 hints that, at least in this case, GP has got stuck and no further improvements will be found.

In Figure 6 we plot the evolution of size up to generation 175 488 (again log scales). Notice the surprisingly good agreement with the a power law we reported in [68] for the first thousand generations.

In [73, 74, 75], assuming no anti-bloat measures, we predicted that tree sizes would grow sub-quadratically, i.e. as a power law, with an exponent ≤ 2.0. This has been repeatedly seen. (It would be interesting if mathematical models or theoretical analysis could explain why GP trees tend to grow at about one

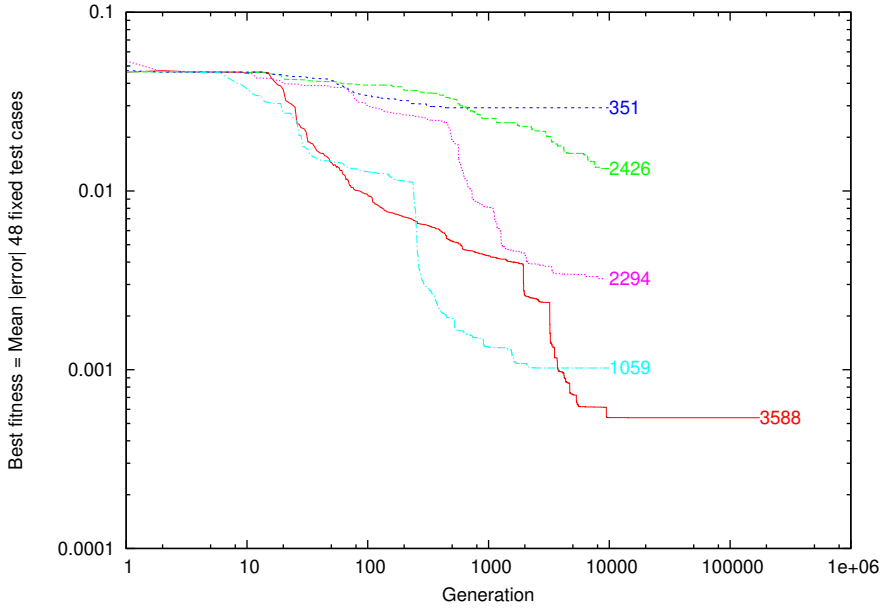

Fig. 5 Evolution of mean absolute error. End of run label gives number of generations when error was reduced. (Note log scales.) Different runs in different colours and line styles. Where possible the same colours and line types are used in all the following graphs.

level increase in *depth* per generation.) In a Boolean problem after evolution for thousands of generations, we saw extreme fitness convergence, causing everyone in selection tournaments to have the same fitness [76] and so loss of selection pressure. This eventually lead to tree size varying at random and no sustained bloat [65,64]. It appears that, with modest population sizes, in the continuous domain (such as symbolic regression) the large number of possible fitness values delays such total fitness convergence. Nonetheless Figure 4 suggests increasing fitness convergence. In even more extend evolution, we might see total loss of selection pressure and with it and end to bloat. However, although in one case the exponent is slightly above 2, so far the prediction of subquadratic bloat is holding (Figure 6).

The mathematics of the distributions of binary trees has been extensively studied [77] [78, page 210]. Average random trees have a somewhat fractal self similar structure in which subtrees within large random trees have shapes which are like those of the random tree they inhabit. Approximately half of large random binary trees have a leaf directly connected to their root node.

A well formed binary tree (of size l) must have a depth d lying between that of a short dumpy tree of the same size (depth= $\lceil \log_2 l \rceil$). And a maximal long stringy tree, consisting of a single run of functions each a leaf as one argument and another function as the other, except the last function, all of whose arguments are leafs, (depth= $(l+1)/2$). Therefore $\log_2 l \leq\ d\ \leq (l+1)/2$. That is, all binary trees must lie between the "Full" and "Tall" lines in Figure 7. In

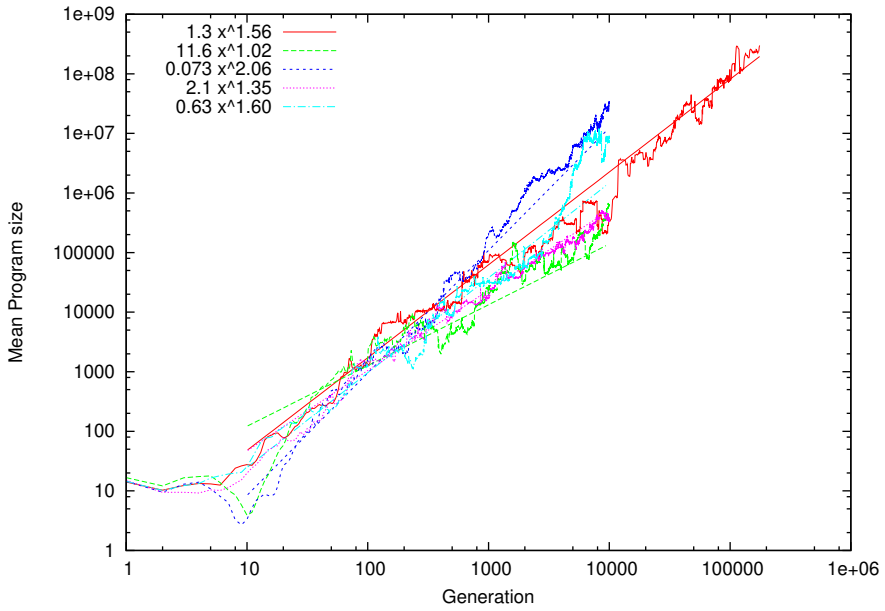

Fig. 6 Evolution of tree size. Straight lines shows best RMS error power law fit between generation 10 and 1000. Colours as Figure 5.

fact most large binary trees lie close to a parabolic curve: depth $\approx\sqrt{2\pi|\text{size}|}$, "Flajolet" in Figure 7 (note log scales). As is expected, in the absence of size or depth limits, even after prolonged evolution, GP trees are somewhat randomly shaped and tend to lie close to Flajolet large tree limit [74].

3 Syntactic Convergence of Tree Shape and Contents

In this and the following sections (up to Section 8) we analyse the evolved populations in much more detailed. Due to space and runtime constraints, this analysis covers the first population created by crossover, i.e. generation 2, and then every 100 generations up to generation 10 000.

Figure 8 shows, despite the wide range in tree sizes in each generation, the population does converge from the root out[8]. The bright yellow indicates the contents (both program element and tree shape) that are identical. Whereas black shows subtrees whose contents or location that are unique. Figure 9 shows populations, with genetic convergence explicitly plotted against depth. Note the non-uniform vertical scale highlights small deviation from perfect convergence. After thousands of generations, hundreds of nested function calls

[8] Figures 8, 15 and 17 use Daida's [40] lattice in which trees are shown with their root at the origin and branches splayed out from the centre using the full 360°. `http://www.cs.ucl.ac.uk/staff/W.Langdon/gp2lattice/gp2lattice.html` This circular display allows populations, indeed multiple generations, of trees to be displayed together, as if plotted on top of each other. It also highlights the asymmetry of the highly evolved trees.

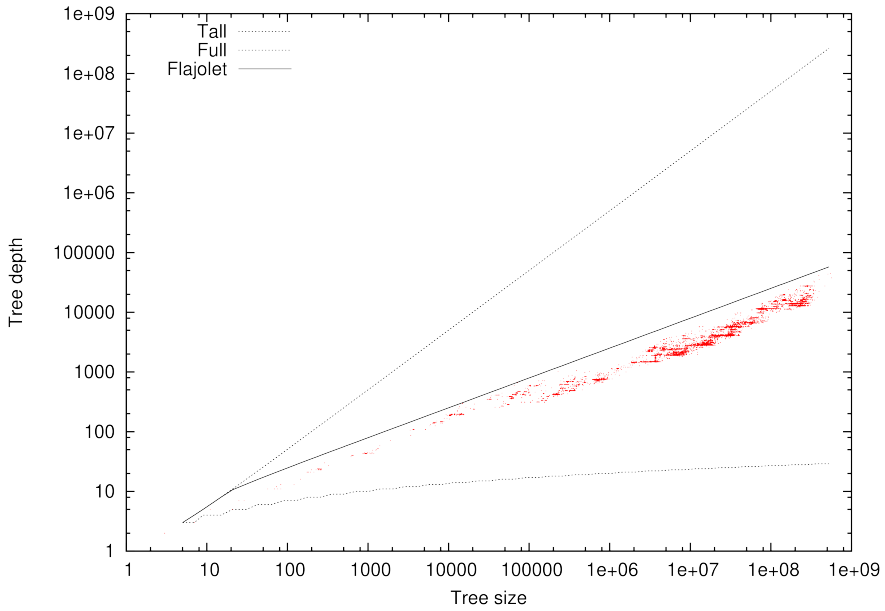

Fig. 7 Plot of size and depth of the best individual in each generation for Sextic polynomial runs with population of 500. Binary trees must lie between short fat trees (lower curve "Full") and "Tall" stringy trees. Most trees are randomly shaped and lie near the Flajolet limit (depth $\approx\sqrt{2\pi|\text{size}|}$, solid line, note log-log scales).

from the root node, there are functions or leafs where most members of the whole population are identical. Turning this around (dashed lines in Figures 10 and 11) there are no unique root nodes and the fraction of unique genes rises to a maximum of only 5.4 parts per million in the population at depth 420. The distribution of unique genes (i.e. program elements and their location in the whole tree) approximately follows the distribution of tree locations in the whole population (Figure 10).

In earlier work on a Boolean problem [65,64], we reported tree distributions were similar to but significantly different from those predicted by Poli and Dignum [79] for crossover only GP without fitness selection. Poli's result holds regardless of problem domain. It appears the initial deviation from the Lagrange distributions predicted by Poli is caused by fitness selection and its continuation during evolution may be due to a founder effect, whereby in a finite population evolution's long term path is influenced by fit individuals from many generations ago. It would be interesting to reconcile Poli's predictions (which seem related to our prediction of random tree shapes) with actual GP runs.

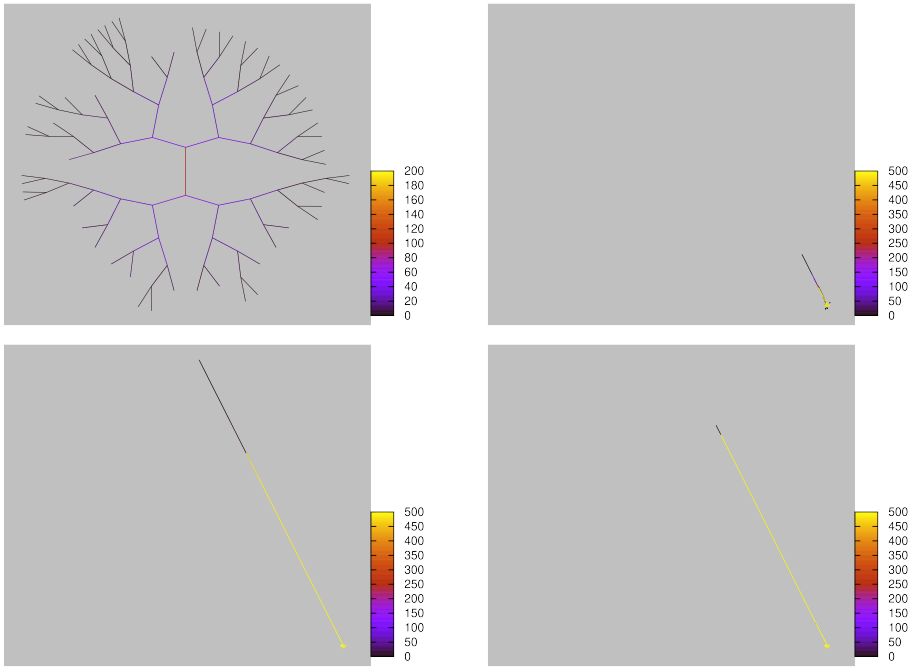

Fig. 8 Genetic convergence in a population of 500 binary trees after 2, 100, 1000 and 10 000 generations (maximum depth 7, 118, 656 and 507). Note 100, 1000 and 10 000 gens at the same scales, but gen 2 expanded to show detail around the origin. Video online: `https://youtu.be/TssAIo-vatE` Trees are plotted in lattice form with their root at the origin 0,0 and branches spread out radially (see especially top left, generation 2). The whole population is plotted with their roots aligned. The colour indicates the degree of convergence at that point in the tree. In Generation 2 less than half the population has identical components but this grows from the root outwards (bright yellow). Darker colours (black) indicate code which only occurs in a few trees or is unique.

4 Semantic Convergence

Although we see some convergence of values returned via the root node of the GP trees, see Figures 12 and 13, it is not as extreme as the genetic convergence around the root nodes. Figure 12 shows the population evolving step wise towards zero error. Although the spread of fitness values is small, typically less than half the population have identical fitness. That is, the small halo of non-converged, diverse, program element far from their root, still have some impact and cause some limited spreading of the values returned by the population's root nodes, even though the halo's values pass through hundreds of functions which are identical across at least 99.6% of the population. This is slightly different from the loss of entropy seen in traditional hand coded programs, as the halo of diversity represents many perturbations around the converged region, rather than a single mutation or error injection point. Nonetheless, the observed evolution of very close fitness values at the root nodes (see right hand end of Figure 13), despite large values present in the evolving trees

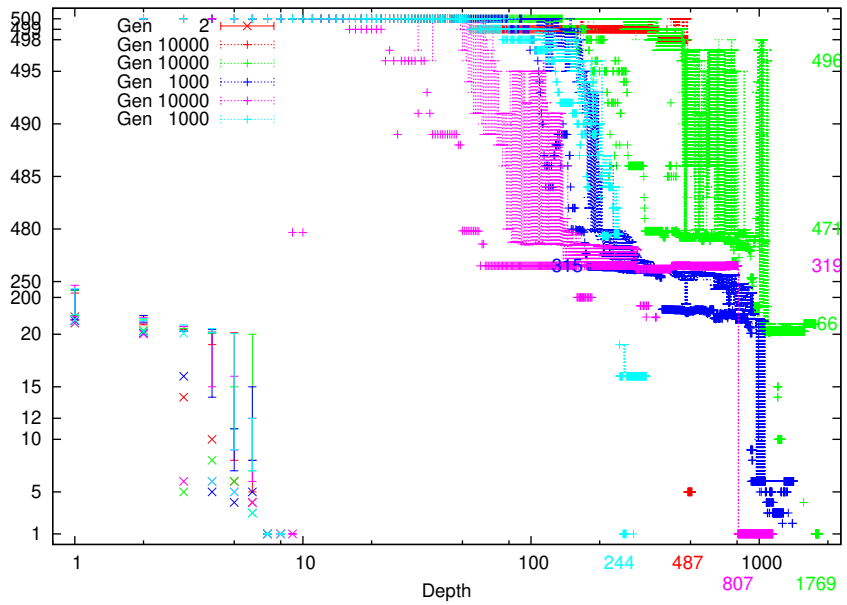

Fig. 9 Genetic convergence of contents and shape of 500 GP trees in five runs at generations 2, 1000 or 10 000. Error bars show interquartile range around the median. Note in later generations the whole population agrees around the root. After thousands of generations most trees have some genetic diversity only more than 200 nested function calls deep (see also Figure 10). Note non-uniform scales.

(see Section 4.1) suggests strong perturbation damping in the large evolved converged nested functions between the halo and the roots.

4.1 Convergence of Values within the Programs

Each function within each tree in the population returns a value for each test case (48). To summarise these 48 values the fitness function is applied not just at the root node but at every function in the tree. Giving a floating point "fitness value" for every executed subtree. (The fraction of non-executed code is dealt with in the next section, Section 5.)

We look at the spread of fitness values across the population at the same points in each tree (inter quartile range vs. median) for every tree location in the population. For the case of the first generation created by crossover (generation 2), the widest normalised spread 3.0 occurs in a function at depth 4. In only 28% of the generation 2 functions is the interquartile spread less than the median. In contrast, in generation 10 000 94.8% of functions have identical subfitness.

Despite the use of protected division, to prevent division by zero, the huge numerical expressions evolved cause floating point numerical overflow (-inf) and subfitness values as small as $-2.11\ 10^{36}$ (almost -FLT_MAX/48).

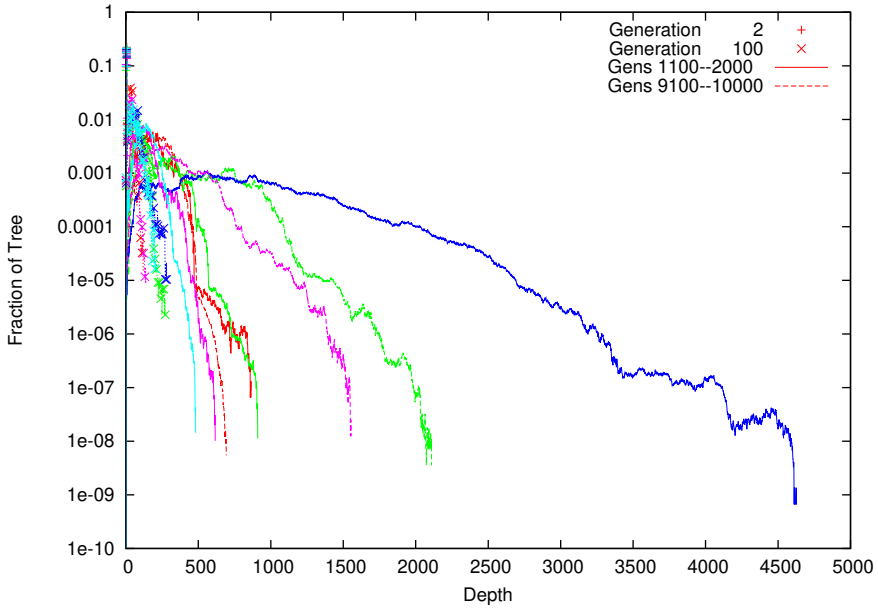

Fig. 10 Distribution of nodes within population of 500 GP trees by distance from root node. Note log vertical scale. To smooth data (and be compatible with Figure 11) two longer traces are the means of ten samples over 1000 generations. Note approximate Gaussian shape, albeit with long tails. Colours as Figure 5.

Figure 14 shows the evolution of phenotypic convergence in terms of the population ratio of inter quartile range to median. In almost all cases the population spread is less than 1000^{th} of the median value. Indeed by generation 10 000, 95% of syntactically converged functions calculate exactly the same value and in 99.9% the population spread is less than 100^{th} of the median value. Right of Figure 14 shows the evolution of the tiny fraction of very diverse subfitness calculations.

Figure 15 shows the distribution of phenotypic convergence across the trees using Daida's [40] lattice format for the complete population. As with genetic convergence phenotypic convergence grows outwards from the root node. (A YouTube video shows the evolution of phenotypic convergence, as a movie `https://youtu.be/_qz1_1AK1gw`)

5 Negative Results Sometimes Too Few Introns

It is well known that multiplying by zero produces zero regardless of multiply's other argument. Similarly, in "protected" division, division by zero produces one regardless of the other argument. In GPavx we tried improving the speed of the GP by recognising the presence of zero as the first argument of multiply and skipping the evaluation of multiply's second argument. (The same trick can be done with protected division, however this requires the order of the

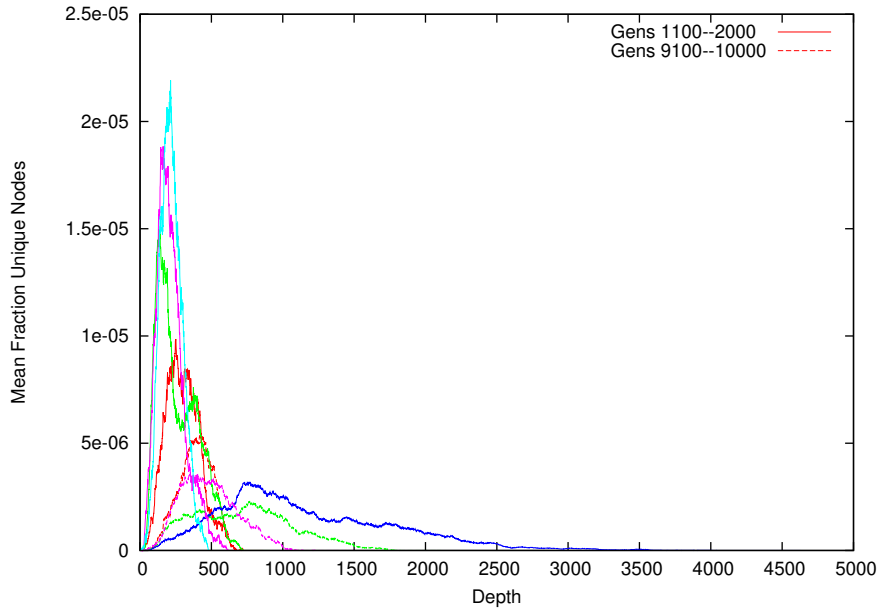

Fig. 11 Distribution of tiny fraction of unique nodes within populations of 500 GP trees by distance from root node (cf. Figure 10). To smooth the plots, the means of ten samples (a hundred generations a part) are shown. Note approximate Gaussian shape. Colours as Figure 5.

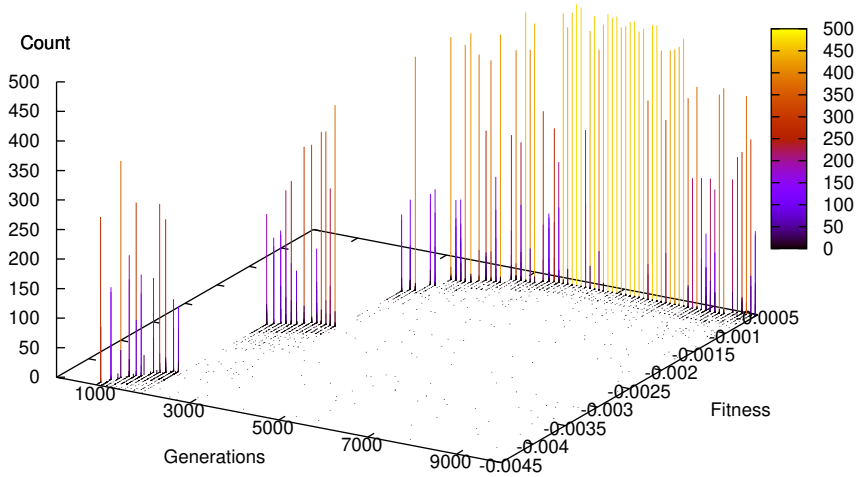

Fig. 12 Distribution of fitness within populations of 500 GP trees. Note many trees have identical fitness (lighter colours), as their convergence around their root nodes (cf. Figure 8) would suggest. But differences far from their root (cf. red + in Figure 9) still cause many differences to the values output at their roots.

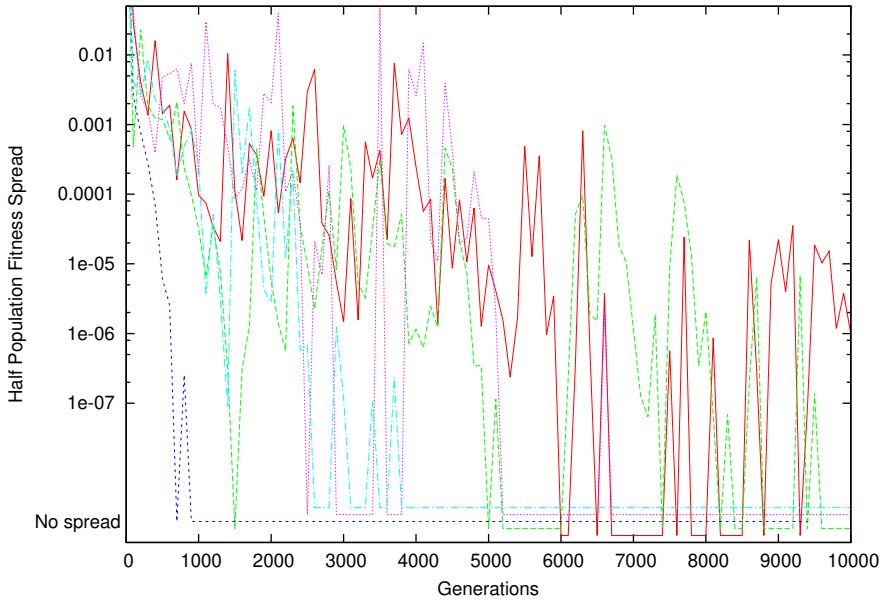

Fig. 13 Convergence of best half of the population fitness range expressed as a fraction of the best fitness. (First run, red line, same data as Figure 12.) "No spread" indicates cases with more than half the population having identical fitness. Colours as Figure 5.

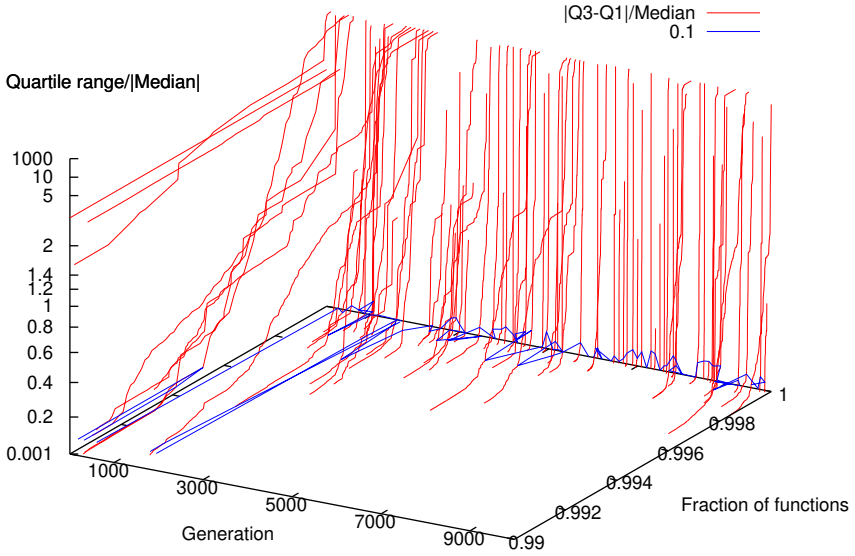

Fig. 14 Evolution of semantic diversity across the same tree positions in the population. After generation 1900 at least 99% of functions are highly converged and have less than 0.1% variation in fitness across the population (values 0.1% below not plotted). Blue contour line at 10%. In generation 10 000 almost 95% of functions always return the same value and for 99.9% of functions the interquartile variation in fitness is less than 1% of their median value. Note non-linear vertical scale (ratio capped at 1000).

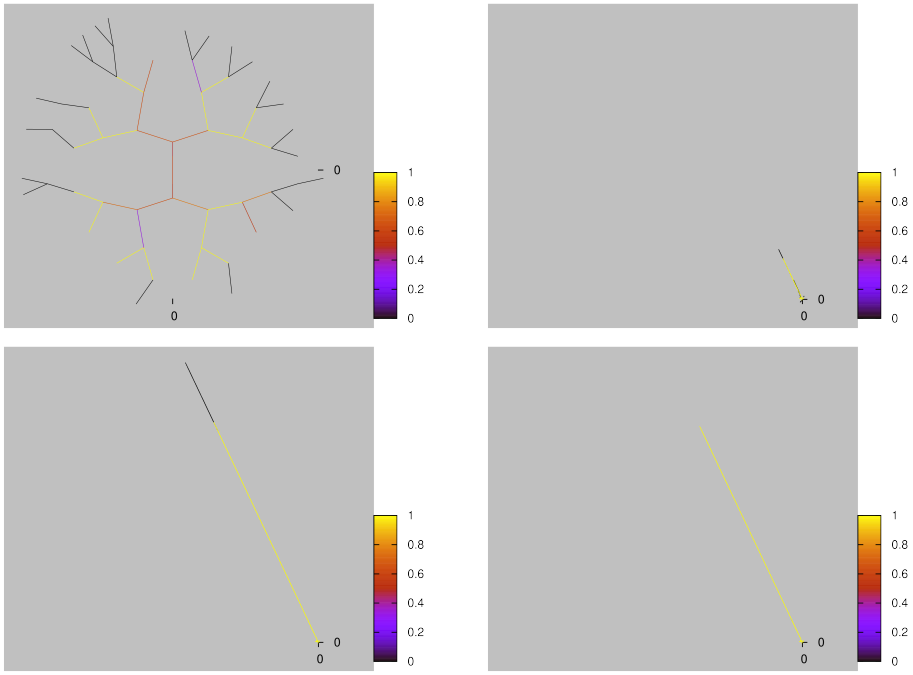

Fig. 15 Phenotypic convergence median/inter quartile range in a population of 500 binary trees after 2, 100, 1000 and 10 000 generations (see Figure 8). Ratio capped at 1, so yellow indicates inter quartile range $\leq$ median. Note 100, 1000 and 10 000 gens at the same scales, but gen 2 expanded to show detail around the origin. Video online: `https://youtu.be/_qz1_1AK1gw`

arguments to be reversed, i.e. the divisor must be evaluated first.) To take advantage of the Intel AVX vector instructions we require zero to appear in all the 48 test cases. Despite the evolution of huge bloated trees in some cases this produced almost no speed up.

The fraction of introns varies between runs. In the first run (solid line in Figure 16) although multiply is common and zero is computed in the evolved trees (e.g. by x-x=0), 0.0 is not common (we are after all trying to evolve non-zero values Figure 2 page 6) and so, as the lowest line in Figure 16 shows, typically more than 99.5% of the evolved code is evaluated. However in another run (upper line), we can avoid executing almost all the code by simply treating multiply by zero as a special case.

6 Distribution of Subtree Fitnesses

Not only does the population evolved to have very little genetic variation (Figure 15) but the values passing through them during evaluation also converge. Figure 17 shows the evolution of the typical sub-fitness values in a typical tree. Although fitness improves markedly (see colour bar and note $\log_{10}$ scale) the

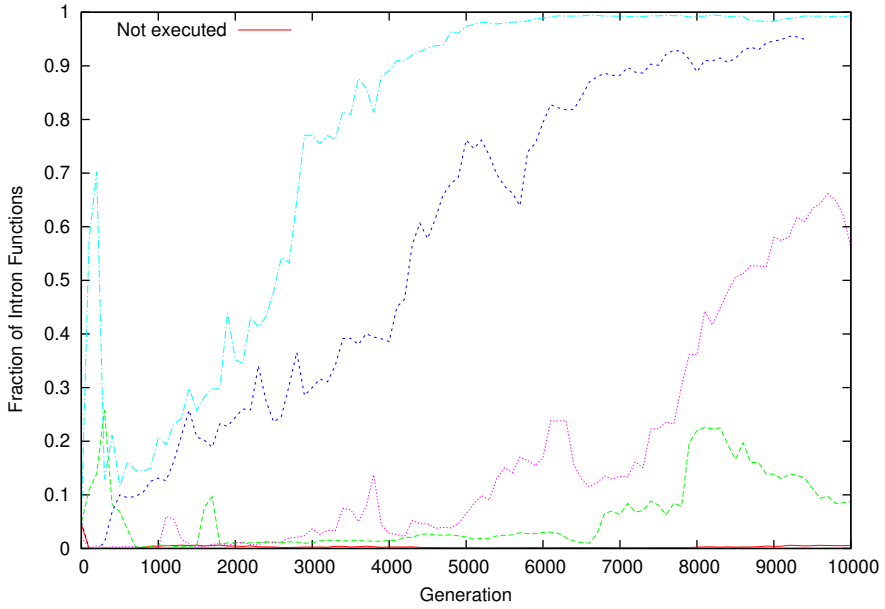

Fig. 16 Evolution of introns. I.e. fraction of functions not executed because they follow multiplication by zero. Colours as Figure 5.

general behaviour is locked-in and evolutionary improvement is channeled via the branches connecting the large subtree to the root node. The channel here typically has relatively poor fitness and so is shown in blue.

Figures 18 and 19 show the values calculated by functions within the populations. In the first run in only two functions do the huge expressions overflow floating point arithmetic to give -inf, nonetheless the wide range of values can be seen. Figure 18 highlights the long term phenotypic stability of the evolving populations, showing similar pattern of values calculated over thousands of generations. It seems the expected highly repetitive nature of trees evolved by crossover [80] is responsible for the enormous number of times certain values are calculated. E.g. 9% of function calculations have a subfitness of -0.509019 (the subfitness of `SUB -0.129 X`). $\frac{1}{3}$ of the functions return just 12 values. Other runs lock into other phenotypic values, Figure 19.

7 Information Flow and Entropy during Execution

Conventional tree GP can be readily thought of in terms of information flow and the entropy of value distributions within the tree. This detailed examination of GP trees as executable functions gives another view of how offspring trees can retain their mother's high fitness by returning the same values as her and so how populations of programs can converge despite continual syntactic modification. We can think of information flowing in from the leafs, moving though the tree and exiting at the root node. Each node within the tree takes

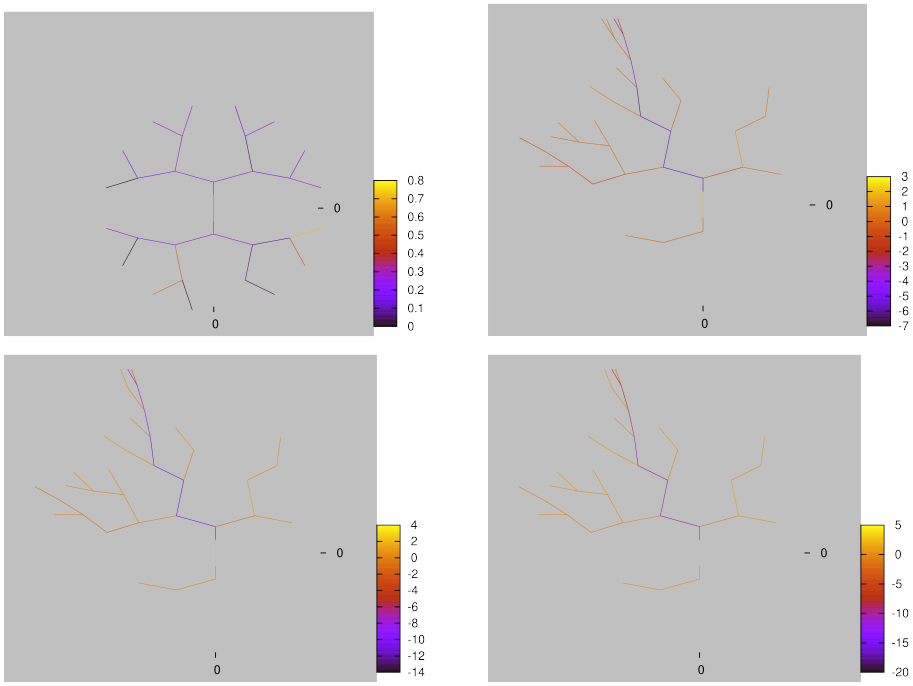

Fig. 17 Evolution of consensus subfitness ($-\log_{10}|\text{median}|$) near the root in a population of 500 binary trees after 2, 100, 1000 and 10 000 generations (see Figure 15). Note bright center at root, where true fitness is calculated. After generation 2, only parts of trees represented by 490 members of the population plotted. (For gen 2 more than 10.)

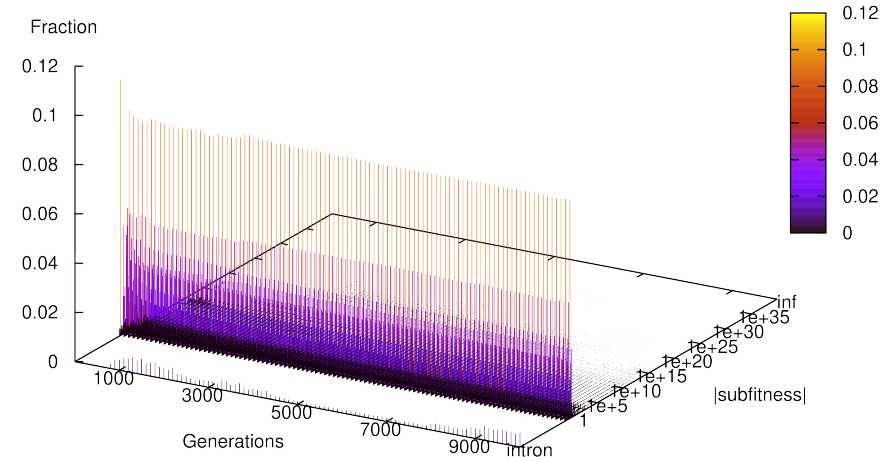

Fig. 18 Evolution of distribution of subfitness within populations of 500 GP trees. (Note log horizontal scale, generation 10 000 detailed in Figure 19 with solid lines.)

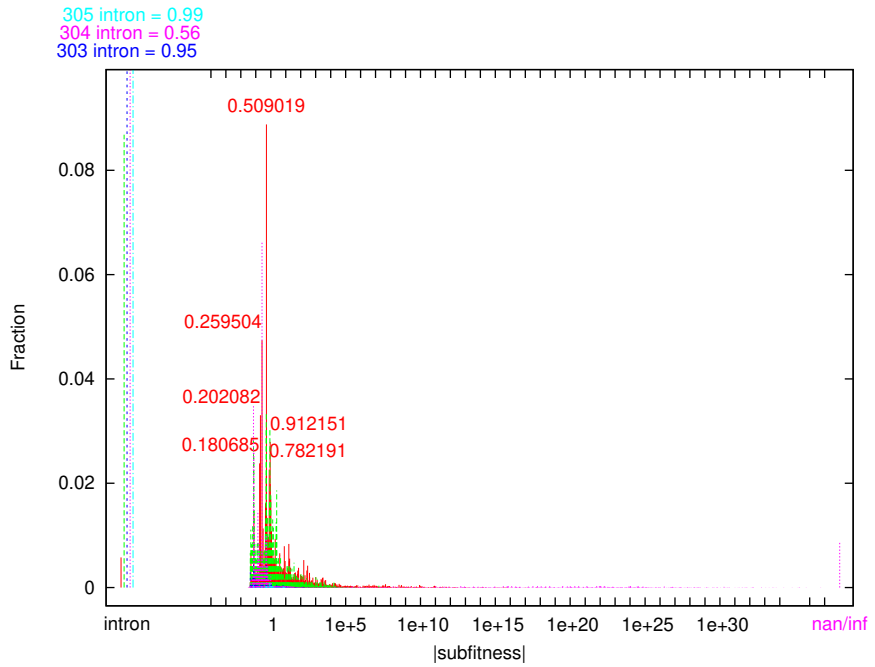

Fig. 19 Distribution of values calculated by functions in generation 10 000. Values in red refer to first run, subfitness 0.509019 is the most common (see Figure 18). The 12 most common values together represent a third of all the functions in the population. Values representing less than 0.1% of the population are group into $1/8^{\text{th}}$ bins. Note despite variation between runs, some subtree values are much more common than others. (Note log horizontal scale.) Colours as Figure 5.

information from two incoming paths (its arguments) transforms it (using its inherent mathematical operation) and then passes the transformed information towards the root node. At the root node the information is gathered via the fitness function into a final value of the GP individual.

As is common in programming, our functions are not reversible [81]. This means, given a function's output, it is impossible in general to infer what its inputs were. E.g. if ADD(x,y) returned 3.1 (without additional information) we cannot say what x or y where. E.g. x,y could be 2.1,1, 2.2,0.9, 2.3,0.8, 1003.1,-1000, etc., etc. That is, we can think of functions as destroying information. Similarly we can think of the fitness function (as it too is irreversible) as also destroying information. For example, here the fitness function takes 48 float values and condenses them to a single float error value[9].

[9] In pretty much any GP system the huge growth with tree size in the number of possible tree shapes (the Catalan number), and the exponential rise in the number of ways of labeling each tree with functions from the function set and leafs from the terminal set, means there must be multiple GP trees with the same fitness value. For example, in our sextic polynomial representation, the number of trees with up to 7 nodes is $1.25\ 10^{12}$, whilst the number of fitness values cannot exceed $2^{32} = 4.29\ 10^{9}$.

In digital computing entropy is usually defined as H=$-\sum_i p_i \log_2(p_i)$ where p_i is the probability of the system being in state i and the summation is over all possible states. It can also be thought of as the expected value of the unexpectedness of the event that the system is in state i ($\log_2(\frac{1}{p_i})$) [82, page 27]. Multiplying by the probability and summing over all possible states to calculate the expected value, gives an equivalent formula H=$\sum_i p_i \log_2(\frac{1}{p_i})$.

In the case of a simple evolved GP tree, the possible states are easy to define and fixed. For example a constant leaf has exactly one state, its value, e.g., -0.995, whereas the leaf x takes one of the 48 fixed unique values -0.97789 to .979541 (see Figure 2). Thus for a constant H=0 and for x H=$log_2(48) = 5.58$ (Notice that the GP uses 48×32 bits = 1536 bits to store the values calculated by evaluating leaf x and $1536 \gg 5.58$, meaning that the representation is highly redundant.) Since there are no side effects, the tree cannot create information, meaning at every node in the GP tree, entropy will be between 0 and 5.58 bits.

In an ordinary programming language we would expect each function to destroy information meaning that entropy would fall monotonically from the start of the program to its end. However, although we start with random expressions, at every generation we are selecting for programs that match the target function as accurately as possible (Figure 2). Thus any GP individual with less than 48 distinct values at its root node will suffer a severe fitness penalty and is liable not to have children. The action of fitness selection with crossover means functions inside the GP trees tend to use x leafs to generate 48 distinct values and trees evolve under our fitness function (which tends to maximise entropy[10]). Therefore evolution tries to ensure that much of each tree has maximum entropy. This is confirmed by Figure 20.

Figure 20 shows in the first population created by crossover, generation 2 (left most violet line), more than half the functions for which both inputs are leafs (i.e. minimum depth=1) calculate 48 different values. In highly evolved trees this climbs to 68%. After generation 100 this reaches 100% for parts of the tree at least four nested function calls from a leaf. That is, apart from near their leafs, GP trees evolve to have the maximum possible entropy (5.58 bits) during fitness execution.

8 Evolution of Phenotypically Robust Code

The fraction of children which are syntactically identical to their parent (because by chance crossover over writes a subtree with an identical one) remains about 5.58% throughout the run. Although some identical subtrees inserted by crossover have three of more nodes, most are caused by replacing a leaf by the same leaf. In these large trees half the tree is composed of leaves, even after prolonged evolution the fraction of x leafs remains near that in the initial

[10] We could even imagine entropy as being a secondary fitness objective, e.g. to penalise initial random trees which calculate fixed values near the average of the target function, rather than trying to match its variation.

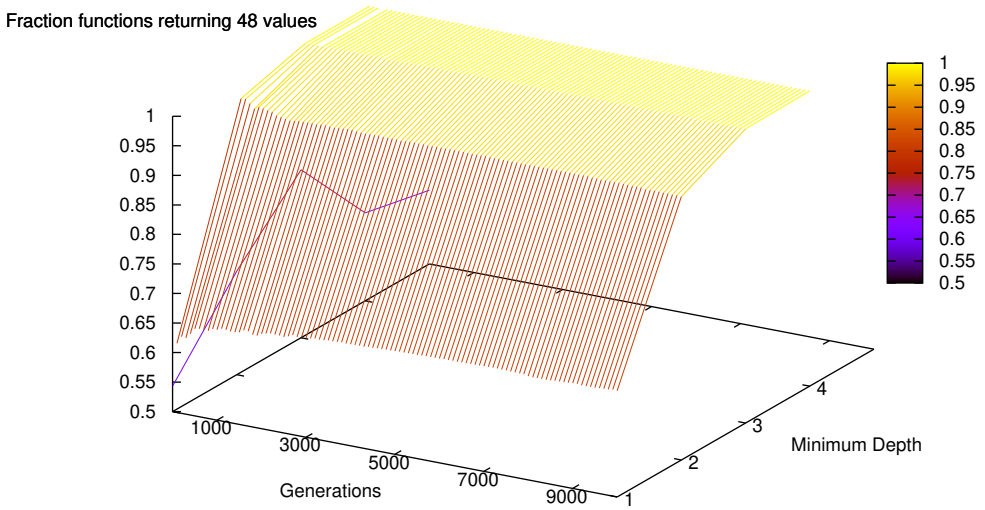

Fig. 20 Evolution of average distribution of maximum (48) possible different evaluation values in GP trees. Y-axis is minimum distance to a leaf.

population, i.e. about 50% of the leafs (25% of all the tree). So the observed fraction of non-syntax disrupting crossovers is approximately the same as the fraction expected in random trees from swapping an x leaf with another x leaf ($6.25\% = 25\% \times 25\%$).

It is easy to check if the inserted subtree is identical to the subtree it replaces. If it is, then the offspring is identical to the parent and, since the fitness function is static, we can skip evaluating it and simply set the child's fitness to be identical to the parent from which it inherited its root node (mum).

Since we are dealing with trees without side effects, they can be evaluated in any order. For example, evaluation from the leafs to the root will give the same result as the usual recursive depth first evaluation from root to the leafs[11].

It is clear that most crossover subtrees are neither genetically nor phenotypically identical. Nonetheless (due to having only pure functions, i.e. with out of side effects) we can independently evaluate both the old and the inserted subtrees. In both cases we will get a vector of floats, each with an element per test case. (In our case the vectors are 48 elements long.) If the two vectors are identical, then the evaluation of the whole of the rest of the child must be identical to that of its parent. And in particular, their fitness must be the same. So, as with genetically identical crossovers, if there is no phenotypic disruption at the crossover point, then there is no phenotypic disruption in the rest of the tree and we can skip the rest of the child's evaluation and simply set its fitness to be the same as its parent. Notice, since the child is identical to

[11] Where recursion is not well supported, interpreting trees as reverse polish expressions can be efficient [83,84].

the parent except below the crossover point, we only need the parent's fitness and the old crossover subtree. Despite the enormous changes to the population during evolution, the fraction of children which are not genetically identical but are phenotypically identical at the crossover point, remains near 0.04%. This means that, by itself, checking for phenotypic convergence of different crossover subtrees can make little difference overall to GP run time. However it is cheap to include more checks for phenotypic convergence.

To check for phenotypic convergence we need only two vectors (of length 48, i.e. 192 bytes each). If the child is phenotypically different from its parent at the crossover point, unless we are at the root node, we can move up one level in the tree towards the root and check again. At the next level up, before we can evaluate the function, we evaluate its other argument, i.e. the other subtree. This avoids saving the complete phenotypic state for every tree (although this can be done somewhat efficiently [85,86]). Notice in the child and the parent, the other subtree will be identical and so it need only be evaluated once. This gives us a third vector. Using these three vectors, we can now evaluate the function above the crossover point for both the child and for the parent, giving two result vectors. If they are the same, everything else in the child is not only genetically identical to the parent but also phenotypically identical and so the child's fitness must be identical to that of its parent and we can stop the evaluation and simply copy the parent's fitness value. Obviously we can repeat this for the next level up, until we either find a level where the child phenotypic disruption caused by a random crossover event has died away to nothing or we reach the root node.

If we find phenotypic convergence between the child and parent then we have saved evaluating part of the child. The best case saving is almost |size|. If we do not, then, in theory, we have paid a small price by evaluating every node on the path from the crossover point to the root node twice (once for the child and again for the parent). As the evolved trees are approximately randomly shaped, on average the length of the path from crossover point to the root will be $\approx 0.5\sqrt{2\pi|\text{size}|}$. I.e., in theory the worst case fractional overhead is only $\approx\sqrt{\frac{\pi/2}{|\text{size}|}}$ (see dotted lines in Figure 22). However in practice repeatedly using EVAL on subtrees, rather than once on the whole child, is liable to be inefficient.

Figure 21 shows initially most crossovers lead to phenotypic disruption throughout the offspring (i.e. from crossover point all the way to the root). However as the population converges and the trees get deeper the fraction falls. Over the first 10 000 generations, on average in between 3% and 19% of children the evaluation cannot be short cut in this way. Depending upon run, there is a clear downward trend so it appears from Figure 21 that the fraction will fall to near zero after prolonged evolution. Indeed Figure 4, page 7, suggests that as the trees become substantially deeper the fraction of phenotypically destructive crossovers does continue to fall. On average, over the last 1000 generations of the longest run, only 0.025% of crossovers disrupt fitness. This hints that, provided the penalty of calling EVAL multiple times is not too big,

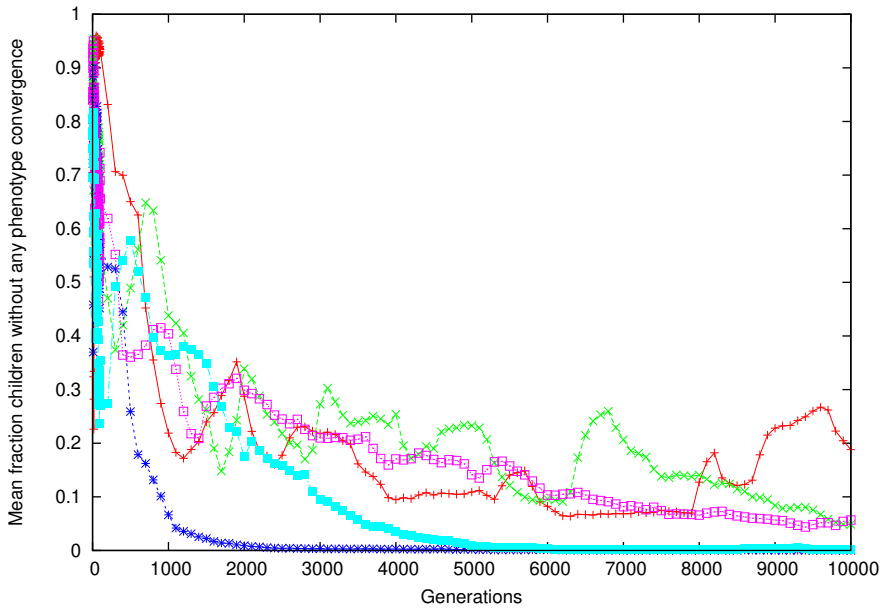

Fig. 21 Evolution of lack of any internal phenotypic convergence. Initially most trees must be fully evaluated but this falls, so over the first 10 000 generation between 81% and 97% of evaluations can be short cut. Data smoothed by plotting running means over 100 generations. Colours as Figure 5.

and so by taking advantage of internal phenotypic convergence between the crossover point and the root, huge evaluation time savings are possible [87, 88]. Figure 22 again concentrates upon the first 10 000 generations.

Notice that Figure 22 is telling us that, later in the runs, the phenotypic disruption caused by crossover almost never reaches the root node.

If we look at this disruption in software engineering terms using Voas' [89] PIE (Propagation, Infection, Execution) model of software bugs, we can view the tree as a program. The whole program (the whole GP tree) is "Executed". Its output is the value returned by the root (the outer most) node. We then view subtree crossover as almost always changing the tree and so injecting an error ("Infecting" the source code). Almost all such errors start to "Propagate" up the tree towards the program's end (the root node). However in large trees, most crossover points lie far from the root node and the "error" (semantic change) must pass through many floating point arithmetic functions to have an impact on the program's final value (the output of the root node). Most functions do propagate the error. But every so often, on a given test case, a function lying between the crossover point (the error) and the root will return the same value as it did on that fitness test case as it did in the parent, i.e. before the crossover. When this happens, above the function the error is no longer visible on that test case. Indeed the error does not reach the program's output (for that test case). What we see, in highly evolved GP trees, is it is highly likely that the error will vanish on all 48 test cases.

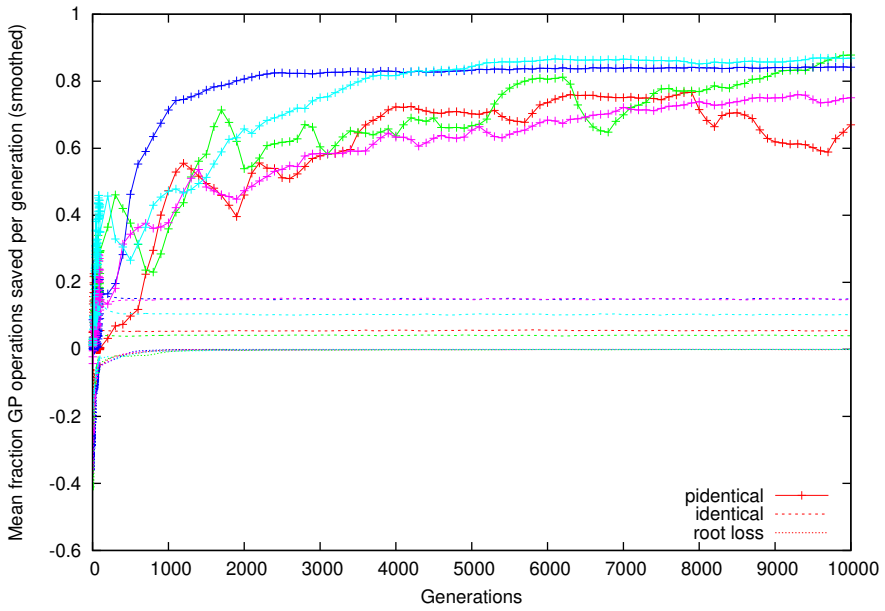

Fig. 22 Evolution of potential savings (>0) in evaluation of GP opcodes per generation by exploiting phenotypic convergence between child and parent. Penalty, <0 dotted, shows after generation 100 for all 48 fitness test cases 95–99.999998% of all crossover's have no impact. After generation 100, data smoothed by plotting running means of 100 generations. Colours as Figure 5.

Although we have seen this for the four arithmetic operations and evolved GP trees, we expect (due to information loss) [90], that this is a general phenomenon in side effect free digital computing.

9 Limitations

Whilst we have investigated many aspects of convergence in GP, it is not possible to fully answer them all. This section discusses some of these important questions, speculates on their answers, preempts a very initial experiment and speculates on the outcome of possible future experiments.

9.1 Impact of Bloat Prevention

The growth of solution size without commensurate increase in fitness (bloat) has been repeatedly investigated and numerous solution proposed. A common approach is to limit the depth of the evolved trees. In earlier experiments [74] we reported that the impact of a limit on tree depth can be to drive the population from initially straggly randomly shaped trees towards larger more bushy trees of near or exactly the maximum allowed depth. Conversely limiting

the tree size can tend to drive the population towards the denser part of the search space which has many more randomly shaped trees. The most numerous trees of a given size lie near to the "Flajolet" line in Figure 7.

In Figure 7 a depth limit corresponds to a horizontal line. Whilst a size limit corresponds to a vertical line. In the absence of other changes, we can use Figures 6 and 7 to hopefully predict the outcome of imposing either a depth or a size limit. We would expect the population to initially approximately follow Figure 6 with subquadratic growth in size with number of generations. Notice Figures 6 and 7 are plotted with log scales, so there is quite some variation between individual runs. Nonetheless we would expect the population mean tree size in a possible future run to initially lie somewhat near the solid straight line in Figure 6 until it starts to approach the limit (which can be plotted on Figure 7 as a straight line).

In the case of a size limit, we would anticipate the population would evolve so that its mean size lies near (but below) the intersection of the size limit and the Flajolet line. In the case of a depth limit, we would anticipate the population mean to lie close to but below the limit but continue to move to the right. Eventually it might be constrained by the lower limit ("full" trees on Figure 7).

Notice we are implicitly assuming bloated populations, where discovery of improved fitness is rare and many members of the population have the same fitness. Thus fitness plays only a small role, and evolution is actually driven mostly by the interplay of the genetic operators and the geometry of the search space.

A possible confounding factor, is we are assuming the trees are never very small. In a few cases it has been observed [91] that the whole population can converge to a tiny tree (e.g. five nodes), provided it has a relatively high fitness and every self crossover leads to a child of lower fitness. In which case, given moderately strong selection (e.g. tournament selection), the population cannot escape in reasonable time.

In some Boolean experiments [70] we were able to evolve small populations for many generations when everyone in the population had identical fitness for many generations. Under these circumstances there is no fitness driving evolution and instead the population executes a random walk under the genetic operations. Here any biases in the genetic operations will dominate. Subtree crossover has no size bias. However, we would expect any mutation operator with a bias in favour of smaller trees would quickly collapse the population. Conversely we would expect a mutation operator with a positive bias, to continue to drive bloat.

One of the goals of these experiments was to see if a similar limit exists in the continuous domain. We did not find it. My feeling is that it does exist but it requires everyone in the population to have identical fitness for many generations. That is, every crossover must not disrupt fitness. It appears this is only possible when every crossover point is far from the root node. For symbolic regression we could attempt to put some numbers to this. It seems reasonable to assume the trees are randomly shaped, and so a limit on the

number of crossover points near the root can be given by the the mathematics of random trees [77]. For a small number of test cases, we might say we want the chance of any crossover point in 100 generations lying within 400 nodes of the root node to be no more than 50%. Notice we want to limit the number of nodes near the root, which is only weakly related to the distance to furthest node (the tree depth). For a given population size and number of test cases, it might be possible using this approach to estimate the size of trees after which bloat will stop [70].

9.2 Impact of Benchmark Choice

We have intensively studied only one application. Although we have argued in Section 1.6 that the sextic polynomial is typical of symbolic regression, evolution is notorious for throwing up surprises. We cannot say definitively that other problems will behave the same way. For example there are a range of benchmarks will rely on inspection of the tree's contents, require limits on the trees or where the primitives (e.g. move forward) do have side effects, for which at least parts of our analysis will not hold. Nonetheless we expect that it would hold for problems, such as symbolic regression, without limits on the trees or side effects and where fitness is derived from treating the trees as mathematical expressions.

9.3 Impact of Number of Functions

We have evolved trees with only four functions. Although we have argued in Section 1.6 that they are sufficient for symbolic regression, it is often wise in GP experiments to include more functions. Often additional functions are included because they may suit the application. Although we certainly have not demonstrated this; since practical implementations of other mathematical functions also lose information, we would expect in many cases increasing the number of functions would still mean that GP continues to converge in ways similar to those we have seen with a near minimal set of functions.

9.3.1 Support for AVX SIMD Parallel Vector Processing

Intel's AVX functions include square root, exp and log, trigonometric, and hyperbolic functions. Other mathematical functions may be supported via open source AVX libraries, or the GP experimenter may need to roll their own AVX code. Also, by using for loops, scalar functions can be incorporated into our AVX framework (but will lose the parallel processing speed up that AVX provides on some more recent Intel hardware).

9.3.2 Mixed Arity Trees

We have deliberately started with a function set composed of only binary functions as this gives rise to binary trees, which have been extensively studied. However there is work on distributions of trees of mixed arity [77]. Surprisingly the distributions of random mixed arity trees are similar to those of random binary trees. So perhaps future work could carry over our analysis to the mixed arity case.

Our extension of Singleton's GPquick [92] to allow incremental evaluation from the crossover point to the root node, has to navigate up the tree. This navigation currently relies on internal nodes having exactly two lower subtrees. It would need extending to cope with other arities or a mixture of arities. (GPquick allows functions to have 0, i.e. be a leaf, 1, 2, 3 or 4 arguments.)

9.4 Impact of Population Size

From a practical point of view, population size is one of the key GP parameters. A larger population will be more expensive, however if set too small, the population may quickly get trapped by a suboptimal solution. Restarting the GP, rather than continuing an unpromising run may help. Machine resources and the huge size of GP trees we anticipated evolving, necessitated using a modest population size. However both smaller and larger populations are common in GP.

Some mathematical schema models use expected behaviour, i.e. assume an infinite population size, whereas the convergence we have described is in real GP runs with finite populations. In a Boolean problem (rather than floating point symbolic regression), we estimated the final size of the trees would be proportional to the population size [65]. As we argued in Section 9.1, we anticipate this would hold here. That is, we anticipate reducing the population size, will reduce the size of the eventual trees.

Additionally reducing the population size, may mean there is more chance of the population starting from a bad place and converging to a suboptimal solution. For a fixed selection method, reducing the population size, also reduces the "take over time" [2].

9.5 Impact of Tournament Size

The mathematics of tournament selection, has been extensively studied [93, 91]. The relationship between tournament size and selection pressure has been formally derived. However typically it is assumed that the population is itself not evolving when extrapolating forward a few generations to calculate Goldberg's "take over time" [2].

As is common in GP, we have used a fixed tournament size of seven (see Table 1). Given we are using a smallish population size, we anticipate either

increasing or reducing the tournament size, particularly at the start of the run, will change in detail the course of evolution and the performance of the evolved solutions. Nonetheless, we anticipate as the trees grow to enormous size the details of the selection technique will become unimportant and provided there is sufficient selection pressure to prevent lower fitness tree reproducing, convergence will (in the absence of other changes) be similar to that we have found. That is, we anticipate a tournament size of 2 or more will cause similar convergence.

9.6 Impact of Test Set Size

We anticipate changes to the fitness function due to changing the number of test points will lead to changes to the relative fitness of various members of the population. As with the selection mechanism (previous section), particularly at the start of the run, this will lead to differences in the detailed path of evolution and the quality of the solutions found. However again, we anticipate as the trees become large, a change in the number of fitness tests will not in itself make a substantial difference to convergence.

In the case of increasing the strength of the test oracle by increasing the number of tests, we can go a little further. If we have n tests and we assume each test is independent, we can very crudely model our incremental bottom up evaluation. Each step up the tree, we run n tests. Let us assume each test has the same chance p of giving the same result in parent and child. If this happens on a particular test, then that test will yield the same result on mum and offspring on every further node up the tree. We proceed evaluating up the tree until all n tests give identical values in the mum and her child.

This is analogous to the coupon collector problem, except instead of collecting a coupon each turn: 1) we have only a chance p of an individual test giving us a coupon (i.e. mum and offspring evaluating to identical values). 2) We are going to run n tests at once. Given p is fixed, the time taken to first drawing a coupon will follow a geometric distribution with mean $1/p$. If we draw one coupon at a time, the expected time to draw all n coupons is n times the n^{th} harmonic number, so a little more than $n \log n$.

Supposing a particular crossover does not change fitness. Since we have to try on average $1/p$ times to get a coupon but we are playing all n tests at each level up the tree, the expected number of levels required before we can stop bottom up incremental evaluation is $\approx n/(np) \log n = \frac{1}{p} \log n$.

That is, the depth of trees needed before crossover tends to stop being disruptive will scale as $\mathrm{O}(\log n)$ and so tree size would scales as $\mathrm{O}((\log n)^2)$.

In a very initial experiment, we saw that if we increased the number of test cases from 48 to 1000 (i.e. by 20.8 fold) then incremental evaluation needed approximately twice as many steps up the trees ($log(1000)/log(48) = 1.78$). We should stress again that these are very crude models and very initial results, nevertheless they tend to suggests convergence will be relatively insensitive to the number of tests in the fitness test set.

10 Conclusions

From an optimisation exploration–exploitation view point, evolution has defeated a disruptive crossover operator to produced highly converged populations, which exploitatively search the local neighbourhood of the best solution without long range exploration. Although here we see tremendous bloat, Section 5 shows not evaluating easily recognised introns can sometimes fail to reduce computational cost overmuch. Convergence makes optimisation slow and expensive, nevertheless, in Section 8, our study has given hints for reducing the computational cost of long term GP experiments.

We have measured GP convergence in multiple ways. In our very simple GP computer system, evolution builds on top of conserved code. In genetic programming the conserved code is around the tree's root node.

Acknowledgements

I would like to thank my anonymous reviewers, Simon Tatham for `PuTTY`, and Dagstuhl Seminars 17191 on the theory of randomized heuristics and 18052 on Genetic Improvement of Software [94], for inspiring conversations.

Funded by EPSRC GGGP and InfoTestSS grants EP/M025853/1, EP/P005888/1.

References

1. Holland, J.H.: Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence. MIT Press (1992), first Published by University of Michigan Press 1975
2. Goldberg, D.E.: Genetic Algorithms in Search Optimization and Machine Learning. Addison-Wesley (1989)
3. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge, MA, USA (1992), `http://mitpress.mit.edu/books/genetic-programming`
4. Poli, R., Langdon, W.B., McPhee, N.F.: A field guide to genetic programming. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk` (2008), `http://www.gp-field-guide.org.uk`, (With contributions by J. R. Koza)
5. Rothlauf, F.: Representations for genetic and evolutionary algorithms. Springer, pub-SV:adr, second edn. (2006), `http://dx.doi.org/10.1007/3-540-32444-5`
6. Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D.: Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications. Morgan Kaufmann, San Francisco, CA, USA (Jan 1998), `https://www.amazon.co.uk/Genetic-Programming-Introduction-Artificial-Intelligence/dp/155860510X`
7. O'Neill, M., Ryan, C.: Grammatical Evolution: Evolutionary Automatic Programming in a Arbitrary Language, Genetic programming, vol. 4. Kluwer Academic Publishers (2003), `http://dx.doi.org/10.1007/978-1-4615-0447-4`
8. Miller, J.F. (ed.): Cartesian Genetic Programming. Natural Computing Series, Springer (2011), `http://dx.doi.org/10.1007/978-3-642-17310-3`
9. Nikolaev, N., Iba, H.: Adaptive Learning of Polynomial Networks Genetic Programming, Backpropagation and Bayesian Methods. No. 4 in Genetic and Evolutionary Computation, Springer (2006), `http://www.springer.com/computer/ai/book/978-0-387-31239-2`, june

10. Krawiec, K.: Behavioral Program Synthesis with Genetic Programming, Studies in Computational Intelligence, vol. 618. Springer International Publishing (2015), http://dx.doi.org/10.1007/978-3-319-27565-9
11. Wong, M.L., Leung, K.S.: Data Mining Using Grammar Based Genetic Programming and Applications, Genetic Programming, vol. 3. Kluwer Academic Publishers (Jan 2000), http://www.springer.com/computer/ai/book/978-0-7923-7746-7
12. Le Goues, C., Pradel, M., Roychoudhury, A.: Automated program repair. Communications of the ACM 62(12), 56–65 (Dec 2019), http://dx.doi.org/10.1145/3318162
13. White, D.R., Arcuri, A., Clark, J.A.: Evolutionary improvement of programs. IEEE Transactions on Evolutionary Computation 15(4), 515–538 (Aug 2011), http://dx.doi.org/10.1109/TEVC.2010.2083669
14. Alexander, B.J., Gratton, M.J.: Constructing an optimisation phase using grammatical evolution. In: Tyrrell, A. (ed.) 2009 IEEE Congress on Evolutionary Computation. pp. 1209–1216. IEEE Computational Intelligence Society, IEEE Press, Trondheim, Norway (18-21 May 2009), http://dx.doi.org/10.1109/CEC.2009.4983083
15. Langdon, W.B.: Genetic improvement of programs. In: Matousek, R. (ed.) 18th International Conference on Soft Computing, MENDEL 2012. Brno University of Technology, Brno, Czech Republic (27-29 Jun 2012), http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/Langdon_2012_mendel.pdf, invited keynote
16. Langdon, W.B.: Genetically improved software. In: Gandomi, A.H., Alavi, A.H., Ryan, C. (eds.) Handbook of Genetic Programming Applications, chap. 8, pp. 181–220. Springer (2015), http://dx.doi.org/10.1007/978-3-319-20883-1_8
17. Langdon, W.B., Harman, M.: Optimising existing software with genetic programming. IEEE Transactions on Evolutionary Computation 19(1), 118–135 (Feb 2015), http://dx.doi.org/10.1109/TEVC.2013.2281544
18. Petke, J., Harman, M., Langdon, W.B., Weimer, W.: Using genetic improvement and code transplants to specialise a C++ program to a problem class. In: Nicolau, M., Krawiec, K., Heywood, M.I., Castelli, M., Garcia-Sanchez, P., Merelo, J.J., Rivas Santos, V.M., Sim, K. (eds.) 17th European Conference on Genetic Programming. LNCS, vol. 8599, pp. 137–149. Springer, Granada, Spain (23-25 Apr 2014), http://dx.doi.org/10.1007/978-3-662-44303-3_12
19. Petke, J.: Constraints: The future of combinatorial interaction testing. In: 2015 IEEE/ACM 8th International Workshop on Search-Based Software Testing. pp. 17–18. Florence (May 2015), http://dx.doi.org/doi:10.1109/SBST.2015.11
20. Petke, J., Harman, M., Langdon, W.B., Weimer, W.: Specialising software for different downstream applications using genetic improvement and code transplantation. IEEE Transactions on Software Engineering 44(6), 574–594 (Jun 2018), http://dx.doi.org/10.1109/TSE.2017.2702606
21. Petke, J., Haraldsson, S.O., Harman, M., Langdon, W.B., White, D.R., Woodward, J.R.: Genetic improvement of software: a comprehensive survey. IEEE Transactions on Evolutionary Computation 22(3), 415–432 (Jun 2018), http://dx.doi.org/doi:10.1109/TEVC.2017.2693219
22. Langdon, W.B., Poli, R.: Foundations of Genetic Programming. Springer-Verlag (2002), http://dx.doi.org/10.1007/978-3-662-04726-2
23. White, D.: An overview of schema theory. arXiv (12 Jan 2014), http://arxiv.org/abs/1401.2651
24. O'Reilly, U.M., Oppacher, F.: The troubling aspects of a building block hypothesis for genetic programming. In: Whitley, L.D., Vose, M.D. (eds.) Foundations of Genetic Algorithms 3. pp. 73–88. Morgan Kaufmann, Estes Park, Colorado, USA (31 Jul–2 Aug 1994), http://dx.doi.org/10.1016/B978-1-55860-356-1.50008-X, published 1995
25. Poli, R., Stephens, C.R., Wright, A.H., Rowe, J.E.: A schema theory based extension of Geiringer's theorem for linear GP and variable length GAs under homologous crossover. In: De Jong, K.A., Poli, R., Rowe, J.E. (eds.) Foundations of Genetic Algorithms VII. pp. 45–62. Morgan Kaufmann, Torremolinos, Spain (4-6 Sep 2002), http://gpbib.cs.ucl.ac.uk/gp-html/poli_2002_foga.html, published 2003
26. Poli, R., McPhee, N.F.: General schema theory for genetic programming with subtree-swapping crossover: Part I. Evolutionary Computation 11(1), 53–66 (Mar 2003), http://dx.doi.org/10.1162/106365603321829005

27. Poli, R., McPhee, N.F.: General schema theory for genetic programming with subtree-swapping crossover: Part II. Evolutionary Computation 11(2), 169–206 (Jun 2003), http://dx.doi.org/10.1162/106365603766646825
28. Poli, R., McPhee, N.F., Rowe, J.E.: Exact schema theory and markov chain models for genetic programming and variable-length genetic algorithms with homologous crossover. Genetic Programming and Evolvable Machines 5(1), 31–70 (Mar 2004), http://dx.doi.org/10.1023/B:GENP.0000017010.41337.a7
29. Poli, R., Vanneschi, L., Langdon, W.B., McPhee, N.F.: Theoretical results in genetic programming: The next ten years? Genetic Programming and Evolvable Machines 11(3/4), 285–320 (Sep 2010), http://dx.doi.org/10.1007/s10710-010-9110-5, tenth Anniversary Issue: Progress in Genetic Programming and Evolvable Machines
30. Holland, J.H.: Genetic algorithms and the optimal allocation of trials. SIAM Journal on Computation 2, 88–105 (1973), http://dx.doi.org/10.1137/0202009
31. Altenberg, L.: The Schema Theorem and Price's Theorem. In: Whitley, L.D., Vose, M.D. (eds.) Foundations of Genetic Algorithms 3. pp. 23–49. Morgan Kaufmann, Estes Park, Colorado, USA (31 Jul–2 Aug 1994), http://dx.doi.org/10.1016/B978-1-55860-356-1.50006-6, published 1995
32. Whigham, P.A.: A schema theorem for context-free grammars. In: 1995 IEEE Conference on Evolutionary Computation. vol. 1, pp. 178–181. IEEE Press, Perth, Australia (29 Nov - 1 Dec 1995), http://dx.doi.org/10.1109/ICEC.1995.489140
33. Rosca, J.P., Ballard, D.H.: Rooted-tree schemata in genetic programming. In: Spector, L., Langdon, W.B., O'Reilly, U.M., Angeline, P.J. (eds.) Advances in Genetic Programming 3, chap. 11, pp. 243–271. MIT Press, Cambridge, MA, USA (Jun 1999), http://www.cs.ucl.ac.uk/staff/W.Langdon/aigp3/ch11.pdf
34. Hoai, N.X.: A Flexible Representation for Genetic Programming from Natural Language Processing. Ph.D. thesis, Australian Defence force Academy, University of New South Wales, Australia (Dec 2004), http://handle.unsw.edu.au/1959.4/38750
35. Greene, W.A.: Schema disruption in chromosomes that are structured as binary trees. In: Deb, K., Poli, R., Banzhaf, W., Beyer, H.G., Burke, E., Darwen, P., Dasgupta, D., Floreano, D., Foster, J., Harman, M., Holland, O., Lanzi, P.L., Spector, L., Tettamanzi, A., Thierens, D., Tyrrell, A. (eds.) Genetic and Evolutionary Computation – GECCO-2004, Part I. Lecture Notes in Computer Science, vol. 3102, pp. 1197–1207. Springer-Verlag, Seattle, WA, USA (26-30 Jun 2004), http://dx.doi.org/10.1007/978-3-540-24854-5_116
36. Zojaji, Z., Ebadzadeh, M.M.: Semantic schema theory for genetic programming. Applied Intelligence 44(1), 67–87 (Jan 2016), http://dx.doi.org/10.1007/s10489-015-0696-4
37. Zojaji, Z., Ebadzadeh, M.M.: An improved semantic schema modeling for genetic programming. Soft Computing 22(10), 3237–3260 (May 2018), http://dx.doi.org/10.1007/s00500-017-2781-6
38. Daida, J.M., Hilss, A.M.: Identifying structural mechanisms in standard genetic programming. In: Cantú-Paz, E., Foster, J.A., Deb, K., Davis, D., Roy, R., O'Reilly, U.M., Beyer, H.G., Standish, R., Kendall, G., Wilson, S., Harman, M., Wegener, J., Dasgupta, D., Potter, M.A., Schultz, A.C., Dowsland, K., Jonoska, N., Miller, J. (eds.) Genetic and Evolutionary Computation – GECCO-2003. LNCS, vol. 2724, pp. 1639–1651. Springer-Verlag, Chicago (12-16 Jul 2003), http://dx.doi.org/10.1007/3-540-45110-2_58
39. Daida, J.M., Bertram, R.R., Stanhope, S.A., Khoo, J.C., Chaudhary, S.A., Chaudhri, O.A., Polito II, J.A.: What makes a problem gp-hard? analysis of a tunably difficult problem in genetic programming. Genetic Programming and Evolvable Machines 2(2), 165–191 (Jun 2001), http://dx.doi.org/10.1023/A:1011504414730
40. Daida, J.M., Hilss, A.M., Ward, D.J., Long, S.L.: Visualizing tree structures in genetic programming. Genetic Programming and Evolvable Machines 6(1), 79–110 (Mar 2005), http://dx.doi.org/10.1007/s10710-005-7621-2
41. Burke, E.K., Gustafson, S., Kendall, G.: Diversity in genetic programming: An analysis of measures and correlation with fitness. IEEE Transactions on Evolutionary Computation 8(1), 47–62 (Feb 2004), http://dx.doi.org/10.1109/TEVC.2003.819263
42. Poli, R., Page, J.: Solving high-order Boolean parity problems with smooth uniform crossover, sub-machine code GP and demes. Genetic Programming and Evolvable Machines 1(1/2), 37–56 (Apr 2000), http://dx.doi.org/10.1023/A:1010068314282

43. Hansen, J.V.: Genetic programming experiments with standard and homologous crossover methods. Genetic Programming and Evolvable Machines 4(1), 53–66 (Mar 2003), http://dx.doi.org/10.1023/A:1021825110329
44. Moraglio, A., Poli, R.: Geometric landscape of homologous crossover for syntactic trees. In: Proceedings of the 2005 IEEE Congress on Evolutionary Computation (CEC-2005). vol. 1, pp. 427–434. IEEE, Edinburgh (2-4 Sep 2005), http://dx.doi.org/10.1109/CEC.2005.1554715
45. Pawlak, T.P.: Competent Algorithms for Geometric Semantic Genetic Programming. Ph.D. thesis, Poznan University of Technology, Poznan, Poland (21 Sep 2015), http://www.cs.put.poznan.pl/tpawlak/link/?PhD
46. Tinos, R., Whitley, D., Ochoa, G.: A new generalized partition crossover for the traveling salesman problem: Tunneling between local optima. Evolutionary Computation 28(2), 255–288 (2020), http://dx.doi.org/10.1162/evco_a_00254
47. Angeline, P.J.: Subtree crossover: Building block engine or macromutation? In: Koza, J.R., Deb, K., Dorigo, M., Fogel, D.B., Garzon, M., Iba, H., Riolo, R.L. (eds.) Genetic Programming 1997: Proceedings of the Second Annual Conference. pp. 9–17. Morgan Kaufmann, Stanford University, CA, USA (13-16 Jul 1997), http://ncra.ucd.ie/COMP41190/SubtreeXoverBuildingBlockorMacromutation_angeline_gp97.ps
48. Haynes, T.: Phenotypical building blocks for genetic programming. In: Back, T. (ed.) Genetic Algorithms: Proceedings of the Seventh International Conference. pp. 26–33. Morgan Kaufmann, Michigan State University, East Lansing, MI, USA (19-23 Jul 1997), http://gpbib.cs.ucl.ac.uk/gp-html/haynes_1997_pbbGP.html
49. Langdon, W.B., Poli, R.: Why ants are hard. In: Koza, J.R., Banzhaf, W., Chellapilla, K., Deb, K., Dorigo, M., Fogel, D.B., Garzon, M.H., Goldberg, D.E., Iba, H., Riolo, R. (eds.) Genetic Programming 1998: Proceedings of the Third Annual Conference. pp. 193–201. Morgan Kaufmann, University of Wisconsin, Madison, Wisconsin, USA (22-25 Jul 1998), http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/WBL.antspace_gp98.pdf
50. Langdon, W.B., Veerapen, N., Ochoa, G.: Visualising the search landscape of the triangle program. In: Castelli, M., McDermott, J., Sekanina, L. (eds.) EuroGP 2017. LNCS, vol. 10196, pp. 96–113. Springer, Amsterdam (19-21 Apr 2017), http://dx.doi.org/10.1007/978-3-319-55696-3_7
51. Ryan, C., Majeed, H., Azad, A.: A competitive building block hypothesis. In: Deb, K., Poli, R., Banzhaf, W., Beyer, H.G., Burke, E., Darwen, P., Dasgupta, D., Floreano, D., Foster, J., Harman, M., Holland, O., Lanzi, P.L., Spector, L., Tettamanzi, A., Thierens, D., Tyrrell, A. (eds.) Genetic and Evolutionary Computation – GECCO-2004, Part II. Lecture Notes in Computer Science, vol. 3103, pp. 654–665. Springer-Verlag, Seattle, WA, USA (26-30 Jun 2004), http://dx.doi.org/10.1007/978-3-540-24855-2_73
52. Sastry, K., O'Reilly, U.M., Goldberg, D.E., Hill, D.: Building block supply in genetic programming. In: Riolo, R.L., Worzel, B. (eds.) Genetic Programming Theory and Practice, chap. 9, pp. 137–154. Kluwer (2003), http://dx.doi.org/10.1007/978-1-4419-8983-3_9
53. Winkler, S.M., Affenzeller, M., Wagner, S.: Analysis of the effects of enhanced selection concepts for genetic programming based structure identification using fine-grained population diversity estimation. In: Krasnogor, N., Lanzi, P.L., Engelbrecht, A., Pelta, D., Gershenson, C., Squillero, G., Freitas, A., Ritchie, M., Preuss, M., Gagne, C., Ong, Y.S., Raidl, G., Gallager, M., Lozano, J., Coello-Coello, C., Silva, D.L., Hansen, N., Meyer-Nieberg, S., Smith, J., Eiben, G., Bernado-Mansilla, E., Browne, W., Spector, L., Yu, T., Clune, J., Hornby, G., Wong, M.L., Collet, P., Gustafson, S., Watson, J.P., Sipper, M., Poulding, S., Ochoa, G., Schoenauer, M., Witt, C., Auger, A. (eds.) GECCO '11: Proceedings of the 13th annual conference companion on Genetic and evolutionary computation. pp. 195–196. ACM, Dublin, Ireland (12-16 Jul 2011), http://dx.doi.org/10.1145/2001858.2001967
54. Langdon, W.B., Poli, R.: An analysis of the MAX problem in genetic programming. In: Koza, J.R., Deb, K., Dorigo, M., Fogel, D.B., Garzon, M., Iba, H., Riolo, R.L. (eds.) Genetic Programming 1997: Proceedings of the Second Annual Conference. pp. 222–230. Morgan Kaufmann, Stanford University, CA, USA (13-16 Jul 1997), http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/WBL.max_gp97.pdf

55. Koetzing, T., Sutton, A.M., Neumann, F., O'Reilly, U.M.: The Max problem revisited: The importance of mutation in genetic programming. Theoretical Computer Science 545, 94–107 (2014), http://dx.doi.org/10.1016/j.tcs.2013.06.014
56. Lissovoi, A., Oliveto, P.S.: Computational complexity analysis of genetic programming. In: Doerr, B., Neumann, F. (eds.) Theory of Evolutionary Computation, chap. 11, pp. 475–518. Natural Computing Series, Springer Nature (2019), http://dx.doi.org/10.1007/978-3-030-29414-4_11
57. Nguyen, A., Urli, T., Wagner, M.: Single- and multi-objective genetic programming: New bounds for weighted order and majority. In: Neumann, F., De Jong, K. (eds.) Foundations of Genetic Algorithms. pp. 161–172. ACM, Adelaide, Australia (16-20 Jan 2013), http://dx.doi.org/10.1145/2460239.2460254
58. Doerr, B., Koetzing, T., Lagodzinski, J.A.G., Lengler, J.: The impact of lexicographic parsimony pressure for ORDER/MAJORITY on the run time. Theoretical Computer Science 816, 144–168 (6 May 2020), http://dx.doi.org/10.1016/j.tcs.2020.01.011
59. Darwin, C.: The Origin of Species. John Murray, penguin classics, 1985 edn. (1859)
60. Owen, R.B., Crossley, R., Johnson, T.C., Tweddle, D., Kornfield, I., Davison, S., Eccles, D.H., Engstrom, D.E.: Major low levels of Lake Malawi and their implications for speciation rates in cichlid fishes. Proceedings of the Royal Society (B) 240(1299), 519–553 (1990), http://www.jstor.org/stable/49477
61. Lenski, R.E., et al.: Sustained fitness gains and variability in fitness trajectories in the long-term evolution experiment with Escherichia coli. Proceedings of the Royal Society B 282(1821) (22 December 2015), http://dx.doi.org/10.1098/rspb.2015.2292
62. Good, B.H., McDonald, M.J., Barrick, J.E., Lenski, R.E., Desai, M.M.: The dynamics of molecular evolution over 60,000 generations. Nature 551, 45—50 (18 October 2017), http://dx.doi.org/doi:10.1038/nature24287
63. McPhee, N.F., Poli, R.: A schema theory analysis of the evolution of size in genetic programming with linear representations. In: Miller, J.F., Tomassini, M., Lanzi, P.L., Ryan, C., Tettamanzi, A.G.B., Langdon, W.B. (eds.) Genetic Programming, Proceedings of EuroGP'2001. LNCS, vol. 2038, pp. 108–125. Springer-Verlag, Lake Como, Italy (18-20 Apr 2001), http://dx.doi.org/10.1007/3-540-45355-5_10
64. Langdon, W.B.: Long-term evolution of genetic programming populations. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion. pp. 235–236. GECCO '17, ACM, Berlin (15-19 Jul 2017), http://dx.doi.org/10.1145/3067695.3075965
65. Langdon, W.B.: Long-term evolution of genetic programming populations. Tech. Rep. RN/17/05, University College, London, London, UK (24 Mar 2017), http://www.cs.ucl.ac.uk/fileadmin/UCL-CS/research/Research_Notes/RN_17_05.pdf, also available as arXiv 1843365
66. Poli, R., Langdon, W.B.: Sub-machine-code genetic programming. In: Spector, L., Langdon, W.B., O'Reilly, U.M., Angeline, P.J. (eds.) Advances in Genetic Programming 3, chap. 13, pp. 301–323. MIT Press, Cambridge, MA, USA (Jun 1999), http://www.cs.ucl.ac.uk/staff/W.Langdon/aigp3/ch13.pdf
67. Langdon, W.B., Soule, T., Poli, R., Foster, J.A.: The evolution of size and shape. In: Spector, L., Langdon, W.B., O'Reilly, U.M., Angeline, P.J. (eds.) Advances in Genetic Programming 3, chap. 8, pp. 163–190. MIT Press, Cambridge, MA, USA (Jun 1999), http://www.cs.ucl.ac.uk/staff/W.Langdon/aigp3/ch08.pdf
68. Langdon, W.B., Banzhaf, W.: Continuous long-term evolution of genetic programming. In: Fuechslin, R. (ed.) Conference on Artificial Life (ALIFE 2019). pp. 388–395. MIT Press, Newcastle (29 Jul - 2 Aug 2019), http://dx.doi.org/10.1162/isal_a_00191
69. Langdon, W.B., Banzhaf, W.: Faster genetic programming GPquick via multicore and advanced vector extensions. Tech. Rep. RN/19/01, University College, London, London, UK (23 Feb 2019), http://www.cs.ucl.ac.uk/fileadmin/user_upload/avx_rn1901.pdf
70. Langdon, W.B.: Parallel GPQUICK. In: Doerr, C. (ed.) GECCO '19: Proceedings of the Genetic and Evolutionary Computation Conference Companion. pp. 63–64. ACM, Prague, Czech Republic (Jul 13-17 2019), http://dx.doi.org/10.1145/3319619.3326770

71. Langdon, W.B.: Genetic improvement of genetic programming. In: Brownlee, A.S., Haraldsson, S.O., Petke, J., Woodward, J.R. (eds.) GI @ CEC 2020 Special Session. p. paper id24061. IEEE Computational Intelligence Society, IEEE Press, internet (19-24 Jul 2020), http://dx.doi.org/10.1109/CEC48606.2020.9185771
72. Langdon, W.B.: Fast generation of big random binary trees. Tech. Rep. RN/20/01, Computer Science, University College, London, Gower Street, London, UK (13 Jan 2020), https://arxiv.org/abs/2001.04505
73. Langdon, W.B.: Linear increase in tree height leads to sub-quadratic bloat. In: Haynes, T., Langdon, W.B., O'Reilly, U.M., Poli, R., Rosca, J. (eds.) Foundations of Genetic Programming. pp. 55–56. Orlando, Florida, USA (13 Jul 1999), http://www.cs.ucl.ac.uk/staff/W.Langdon/fogp/WBL.fogp.ps.gz
74. Langdon, W.B.: Quadratic bloat in genetic programming. In: Whitley, D., Goldberg, D., Cantu-Paz, E., Spector, L., Parmee, I., Beyer, H.G. (eds.) Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000). pp. 451–458. Morgan Kaufmann, Las Vegas, Nevada, USA (10-12 Jul 2000), http://gpbib.cs.ucl.ac.uk/gecco2000/GA069.pdf
75. Langdon, W.B.: Size fair and homologous tree genetic programming crossovers. Genetic Programming and Evolvable Machines 1(1/2), 95–119 (Apr 2000), http://dx.doi.org/10.1023/A:1010024515191
76. Langdon, W.B., Poli, R.: Fitness causes bloat. In: Chawdhry, P.K., Roy, R., Pant, R.K. (eds.) Soft Computing in Engineering Design and Manufacturing. pp. 13–22. Springer-Verlag London (23-27 Jun 1997), http://dx.doi.org/10.1007/978-1-4471-0427-8_2
77. Sedgewick, R., Flajolet, P.: An Introduction to the Analysis of Algorithms. Addison-Wesley (1996)
78. Flajolet, P., Oldyzko, A.: The average height of binary trees and other simple trees. Journal of Computer and System Sciences 25(2), 171–213 (October 1982), https://doi.org/10.1016/0022-0000(82)90004-6
79. Poli, R., Langdon, W.B., Dignum, S.: On the limiting distribution of program sizes in tree-based genetic programming. In: Ebner, M., O'Neill, M., Ekárt, A., Vanneschi, L., Esparcia-Alcázar, A.I. (eds.) Proceedings of the 10th European Conference on Genetic Programming. Lecture Notes in Computer Science, vol. 4445, pp. 193–204. Springer, Valencia, Spain (11-13 Apr 2007), http://dx.doi.org/10.1007/978-3-540-71605-1_18, best paper award
80. Langdon, W.B., Banzhaf, W.: Repeated patterns in genetic programming. Natural Computing 7(4), 589–613 (Dec 2008), http://dx.doi.org/10.1007/s11047-007-9038-8
81. Langdon, W.B.: The distribution of reversible functions is Normal. In: Riolo, R.L., Worzel, B. (eds.) Genetic Programming Theory and Practice, chap. 11, pp. 173–187. Kluwer (2003), http://dx.doi.org/10.1007/978-1-4419-8983-3_11
82. Renyi, A.: A Diary on Information Theory. Probability and Statistics, Applied Probability and Statistics Section, John Wiley and Sons, Chichester (1987)
83. Langdon, W.B., Banzhaf, W.: A SIMD interpreter for genetic programming on GPU graphics cards. In: O'Neill, M., Vanneschi, L., Gustafson, S., Esparcia Alcazar, A.I., De Falco, I., Della Cioppa, A., Tarantino, E. (eds.) Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008. Lecture Notes in Computer Science, vol. 4971, pp. 73–85. Springer, Naples (26-28 Mar 2008), http://dx.doi.org/10.1007/978-3-540-78671-9_7
84. Langdon, W.B., Harrison, A.P.: GP on SPMD parallel graphics hardware for mega bioinformatics data mining. Soft Computing 12(12), 1169–1183 (Oct 2008), http://dx.doi.org/10.1007/s00500-008-0296-x, special Issue on Distributed Bioinspired Algorithms
85. Handley, S.: On the use of a directed acyclic graph to represent a population of computer programs. In: Proceedings of the 1994 IEEE World Congress on Computational Intelligence. vol. 1, pp. 154–159. IEEE Press, Orlando, Florida, USA (27-29 Jun 1994), http://dx.doi.org/10.1109/ICEC.1994.350024
86. McPhee, N.F., Hopper, N.J., Reierson, M.L.: Sutherland: An extensible object-oriented software framework for evolutionary computation. In: Koza, J.R., Banzhaf, W., Chellapilla, K., Deb, K., Dorigo, M., Fogel, D.B., Garzon, M.H., Goldberg, D.E., Iba, H., Riolo, R. (eds.) Genetic Programming 1998: Proceedings of the Third Annual Conference.

p. 241. Morgan Kaufmann, University of Wisconsin, Madison, Wisconsin, USA (22-25 Jul 1998), http://facultypages.morris.umn.edu/~mcphee/Research/Sutherland/sutherland_gp98_announcement.ps.gz
87. Langdon, W.B.: Incremental evaluation in genetic programming. In: Hu, T., Lourenco, N., Medvet, E. (eds.) EuroGP 2021: Proceedings of the 24th European Conference on Genetic Programming. LNCS, vol. 12691, pp. 229–246. Springer Verlag, Virtual Event (7-9 Apr 2021), http://dx.doi.org/10.1007/978-3-030-72812-0_15
88. Langdon, W.B.: Fitness first. In: Banzhaf, W., Trujillo, L., Winkler, S., Worzel, B. (eds.) Genetic Programming Theory and Practice XVIII. Springer, East Lansing, MI, USA (19-21 May 2021), http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/Langdon_2021_GPTP.pdf, forthcoming
89. Voas, J.M.: PIE: a dynamic failure-based technique. IEEE Transactions on Software Engineering 18(8), 717–727 (Aug 1992), http://dx.doi.org/10.1109/32.153381
90. Langdon, W.B., Petke, J., Clark, D.: Dissipative polynomials. In: Veerapen, N., Malan, K., Liefooghe, A., Verel, S., Ochoa, G. (eds.) 5th Workshop on Landscape-Aware Heuristic Search. GECCO 2021 Companion, ACM, Internet (10-14 Jul 2021), http://dx.doi.org/10.1145/3449726.3463147
91. Langdon, W.B.: Data Structures and Genetic Programming. Ph.D. thesis, University College, London, UK (27 Sep 1996), http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/langdon.ps.gz
92. Singleton, A.: Genetic programming with C++. BYTE pp. 171–176 (Feb 1994), http://www.assembla.com/wiki/show/andysgp/GPQuick_Article
93. Blickle, T.: Theory of Evolutionary Algorithms and Application to System Synthesis. Ph.D. thesis, Swiss Federal Institute of Technology, Zurich, Switzerland (Nov 1996), http://dx.doi.org/10.3929/ethz-a-001710359
94. Petke, J., Le Goues, C., Forrest, S., Langdon, W.B.: Genetic improvement of software: Report from dagstuhl seminar 18052. Dagstuhl Reports 8(1), 158–182 (23 Jul 2018), http://dx.doi.org/10.4230/DagRep.8.1.158