Long-Term Evolution Experiment with Genetic Programming

William B. Langdon [1] and Wolfgang Banzhaf [2]

[1] Department of Computer Science, University College London, Gower Street, London WC1E 6BT, UK
[2] Department of Computer Science and Engineering, Michigan State University, East Lansing, USA

Abstract

We evolve floating point Sextic polynomial populations of genetic programming binary trees for up to a million generations. We observe continued innovation but this is limited by their *depth* and suggest deep expressions are resilient to learning as they disperse information, impeding evolvability and the adaptation of highly nested organisms and instead we argue for open complexity. Programs with more than 2 000 000 000 instructions (depth 20 000) are created by crossover. To support unbounded long-term evolution experiments LTEE in GP we use incremental fitness evaluation and both SIMD parallel AVX 512 bit instructions and 16 threads to yield performance *equivalent* to up to 1.1 trillion GP operations per second, 1.1 tera-GPops, on an Intel Xeon Gold 6136 CPU 3.00GHz server.

Keywords

genetic algorithms, GA, genetic programming, GP, convergence, information theory limit on complexity, Long-Term Evolution Experiment, LTEE, extended unlimited evolution, Open Complexity, Speedup technique.

1 Introduction

In evolutionary biology there is discussion about the long-term innovative capabilities of evolution. Some stress that evolution happens on a short time-scale, and even a few hundred generations are enough to produce completely different species (Palumbo, 2001; Owen et al., 1990). Others emphasis that natural evolution is an open-ended process that will continue to produce novelty, even if many millions of generations pass (Evans et al., 2012).

Thus different aspects are considered when studying long-term evolution. One aspect is continuity: If one wants to study evolution in the laboratory, one should strive to set up experiments similar to Nature's evolutionary "experiment" that go on unbroken for a long time. The other aspect is duration: To attempt to evolve for many generations, trusting in the turn-over of information during the evolutionary process. How does evolution proceed after 100, 1000, 10 000 etc. generations of continued evolution? Does it stagnate? Does it continue to produce surprises?

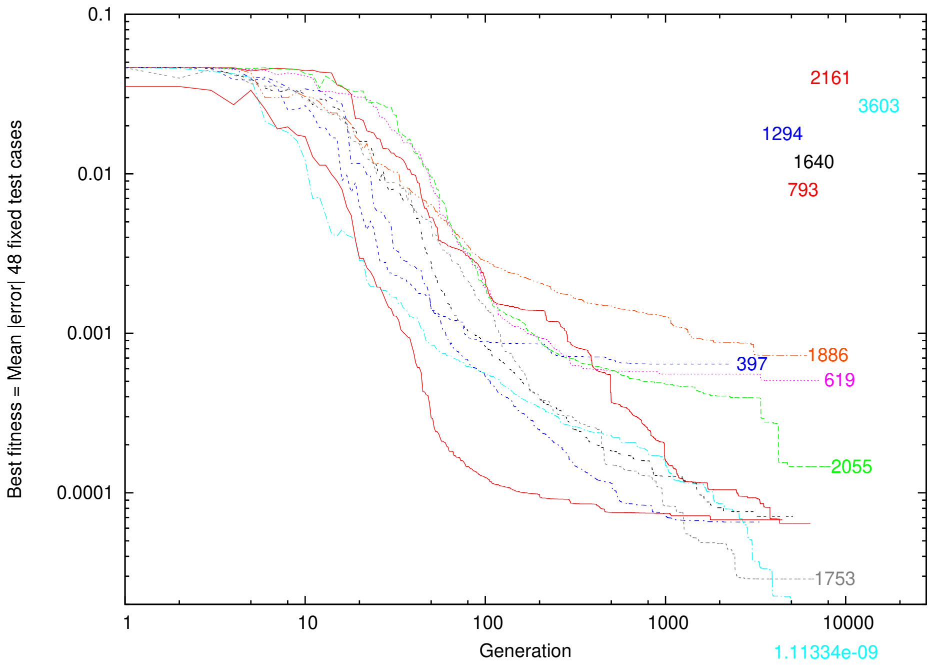

Figure 1: Evolution of mean absolute error in ten runs of Sextic polynomial (Koza, 1992) with population of 4000. (Runs aborted after first crossover to hit 15 million node limit.) End of run label gives number of generations when fitness got better (five shown at top right to avoid crowding).

Richard Lenski and his collaborators have used the evolution of *E. coli* bacterial strains in the laboratory to examine these questions. Since 1988, the evolution of these bacterial strains continues, with the experimental conditions being recorded and bacterial samples being frozen periodically to conserve a time-slice of evolution of these strains (Lenski, 1988). This natural system is studied with both aspects of long-term evolution in mind: The experiment has run uninterrupted since 1988, and the fast reproductive cycle of bacteria allows to study evolution over many generations (Lenski et al., 2015). Already (September 2021) 75 459 generations have been reached, with no end to evolution in sight.

We focus on one aspect of these long-term evolutionary experiments: The number of generations. The medium in which we consider this question, however, is computational. We started to investigate what happens if we allow artificial evolution, specifically genetic programming (GP) with only crossover (Koza, 1992; Banzhaf et al., 1998; Poli et al., 2008), to evolve for tens of thousands, even hundreds of thousands of generations.

With the continuous progress in technology, new hardware has become available, so we built a new GP engine based on Andy Singleton's GPquick (see next section). This allowed us to switch from the Boolean to the continuous domain and run experiments of up to a million generations. At up to 1103 billion GP operations per second (1.1 tera-GPops, see Table 3), this appears to be the fastest single-computer GP system (Langdon, 2013, Tab. 3).

In the Boolean domain often the population quickly found the best possible answer and then retained it exactly for thousands of generations (Langdon, 2017). Nonetheless under subtree crossover we reported interesting population change with trees continuing to evolve. Indeed we were able to report the first signs of an eventual end of bloat due to fitness convergence of the whole population. We can now report in the continuous domain we do see continual innovation and improvement in fitness like in the bacteria experiments. (Note in Lenski's experiments, the E. coli genome contains 4.6 million DNA base pairs.) Figure 1 shows that although the rate of innovation falls, typically better solutions are found even towards the end of the runs. In these runs, there are several hundred or even a few thousand generations where subtree crossover between evolved parents gave a better child.

We are going to run GP far longer than is normally done. Firstly in search of continual evolution but also noting that it is sometimes not safe to extrapolate from the first hundred or so generations. E.g., McPhee (McPhee and Poli, 2001, sect. 1.2) found that his earlier studies which had reported only the first 100 generations could not safely be extrapolated to 3 000 generations.

It must be admitted that without size control we expect bloat[1], and so we need a GP system not only able to run for a million generations[2] but also able to process trees with well in excess of a 100 million nodes[3]. The new system we use is based on Singleton's GPquick (Singleton, 1994; Keith and Martin, 1994; Langdon, 1998), but enhanced to take advantage of both multi-core computing using pthreads and Intel's SIMD AVX parallel floating point operations. Keith and Martin (1994) say GPquick's linearisation of the GP tree will be hard to parallelise. Nevertheless, GPquick was rewritten to use 16 fold Intel AVX-512 instructions to do all operations on each node in the GP tree immediately. Leading to a single eval pass and better cache locality but at the expense of keeping a $T = 48$ wide stack of partial results per thread. In Section 4.2 we deploy a series of speed-up techniques[4] which without changing the course of evolution dramatically speed up GP. Which in turn allowed the evolution of trees five times as big (compare (Langdon and Banzhaf, 2019, Tab. 3) with Table 3) and reduced a run which took a week to one and a half days.

[1] GP's tendency to evolve nonparsimonious solutions has been known since the beginning of genetic programming. E.g. it is mentioned in Jaws (Koza, 1992, page 7). Walter Tackett (Tackett, 1994, page 45) credits Andrew Singleton with the theory that GP bloat is due to the cumulative increase in non-functional code, known as introns. The theory says these protect other parts of the same tree by deflecting genetic operations from the functional code by simply offering more locations for genetic operations. The bigger the introns, the more chance they will be hit by crossover and so the less chance crossover will disrupt the useful part of the tree. Hence bigger trees tend to have children with higher fitness than smaller trees. See also Altenberg (1994); Angeline (1994). Fernandez de Vega et al. (2020) includes a recent summary. In Langdon (2017) we showed prolonged evolution can produce converged populations of functionally identical but genetically different trees comprised of the same central core of functional code next to the root node plus a large amount of variable ineffective sacrificial code.

[2] The median run shown in Figure 2 took 39 hours (mean 62 hours). Under ideal growing conditions, a million generations for E. coli corresponds to about 38 years.

[3] Referring to column 4 in Table 3, we see in two runs crossover creates highly evolved trees containing more than two billion nodes. Note evolution has continued until the size of trees in the population starts to approach INT_MAX. These are by far the largest programs yet evolved.

[4](Langdon and Petke, 2019) (Langdon, 2019) (Langdon, 2020b) (Langdon, 2021b) (Langdon, 2021a) (Langdon, 2022)

Although the populations never lose genetic diversity (Koza's variety)[5], with strong tournament selection (see Table 1) even the larger populations tend to converge to have identical fitness values. However 100% fitness convergence is only seen in long runs with smaller populations (500 or 48 trees). In contrast, in the Boolean domain (Langdon, 2017), even in the bigger populations (500) of that study, there are many generations where the whole population has identical fitness (but again variety is 100%).

The next section describes how GPquick was adapted to take advantage of Intel SIMD instructions able to process 16 floating point numbers in parallel and to use Posix threads to perform crossover and fitness evaluation on 48 cores simultaneously. The Experiments section describes the floating point benchmark (Table 1). Whilst the Results section describes the evolution of fitness and size and depth in populations of 4000, 500 and 48 trees. It finds the earlier predictions of sub-quadratic bloat (Langdon, 1999a) and Flajolet limit (depth $\approx\sqrt{2\pi|\text{size}|}$ (Langdon, 2000b)) to essentially hold. We finish with a short discussion about the continuous evolution permitted by floating point benchmarks and our conclusion that even something as simple as digital evolution in the Sextic polynomial genetic programming benchmark permits continuous innovation.

2 GPquick

First we describe how GPquick is used to do symbolic regression on a simple sixth order polynomial ($y = x^2(x-1)^2(x+1)^2$ known as the Sextic polynomial, Figure 8) and then how GPquick has been modified to run in parallel.

2.1 Sextic and GPquick

Andy Singleton's GPquick (Singleton, 1994) is a well established fast and memory efficient C++ GP framework. In steady state mode (Syswerda, 1990) it stores GP trees in just one byte per tree node. Using separate parent and child populations doubles this (although (Koza et al., 1999) (Langdon, 2020c) shows doubling is not necessary[6]). The 8 bit opcode per tree node allows GPquick to support a number of different functions and inputs. Typically (as in these experiments) the remaining opcodes are used to support about 250 fixed ephemeral random constants (Poli et al., 2008). In the Sextic polynomial we have the traditional four binary floating point operations ($+$, $-$, $\times$ and protected division), an input (x) and 250 constants. The constants are chosen uniformly at random from the 2001 floating point numbers which are multiples of 0.001 between -1 and 1. By chance neither end point nor 0.000 were chosen (see Table 1).

The continuous test cases (x) are selected at random from the interval -1 to +1. At the same time the target value y is calculated (Table 1). Since both x and y are stored in a text file, there may be slight floating point rounding errors due to the standard float⇔string conversions.

[5]Koza defines variety as the percentage of the population that has no genetically identical copy (Koza, 1992, p.93)

[6] To allow the evolution of ultra-large trees, even on a 3TB server, it was necessary to make best use of the available memory. Section 2.4 describes our implementation of (Koza et al., 1999)'s generational GA memory reduction scheme, but for a multi-core computer.

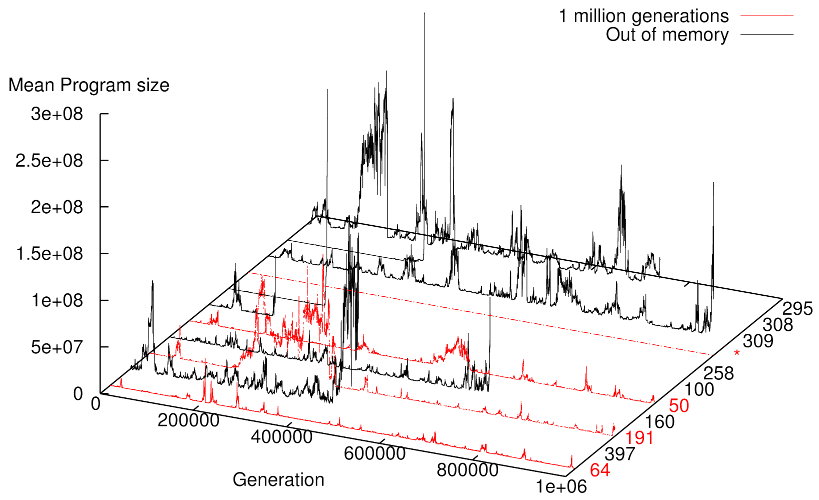

Figure 2: 11 extended runs pop=48. Numbers on right indicate size of largest tree before the run stopped in millions of nodes. One run (*) converged so that more than 90% of the trees contain just five nodes. Three of the other four runs that reached 1 million generations (red) took between half a day and five days. In all but one run (*) we see repeated substantial bloat (> 64 million nodes) and subsequent tree size collapse. Seven runs, in black, terminated due to running out of memory (on server with 46GB).

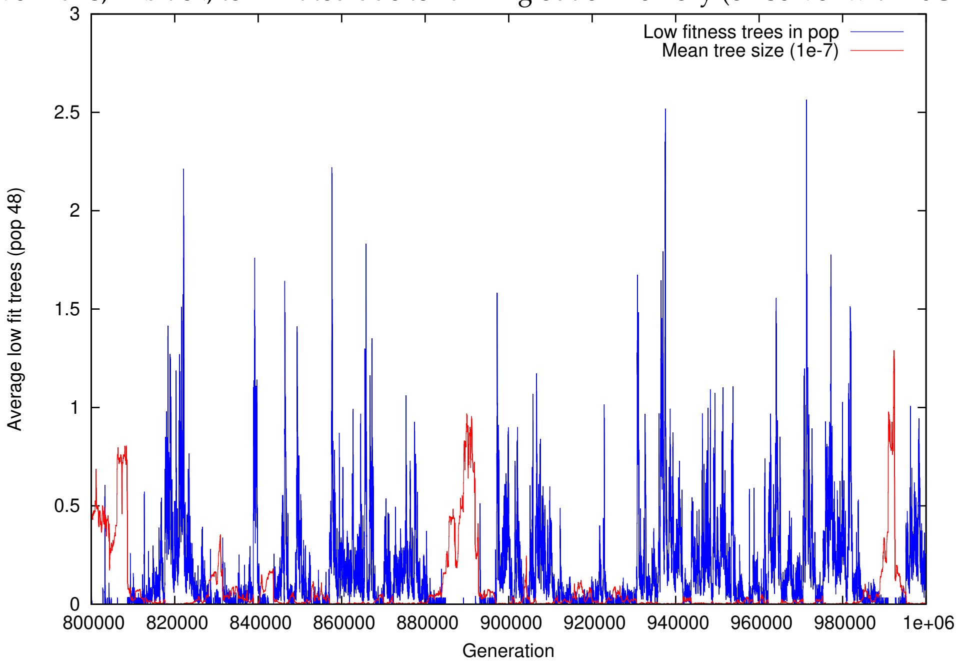

Figure 3: Convergence in last 20% of a typical extended run pop=48. (First run "64" in Figure 2.) Blue curve shows almost all the time there are less than 2 trees without the maximum fitness (smoothed over 30 generations). In 88% of generations plotted the whole population has the same fitness. In 9% all but one have the same fitness and in only 0.6% are there more than two low fitness children. In populations with huge trees (red line), there are many generations where the whole population has identical fitness. Without a fitness differential, tree size may rise or fall.

Whereas the Sextic polynomial is usually solved with 50 test cases (Langdon et al., 1999), since the AVX hardware naturally supports multiples of 16, in our experiments we change this to 48 (i.e. 3×16) (Table 1). The multi-core servers we use each support 48 threads and in the longest extended runs, we reduce the population to 48 (whereas in Langdon (2017) the smallest population considered contained 50 trees).

2.2 AVX GPquick

GPquick stores the GP population by flattening each tree into a linear buffer, with the root node at the start. To avoid heap fragmentation the buffers are all of the same size. The buffer is interpreted once per test case by multiple recursive calls to EVAL and the tree's output is the return value of the outermost EVAL. Each nested EVAL moves the instruction pointer one position forward in the tree's buffer, decodes the opcode there and calls the corresponding function. In the case of inputs x and constants a value is returned via EVAL immediately, whereas ADD, SUB, MUL and DIV will each call EVAL twice to obtain their arguments before operating on them and returning the result. For speed GPquick's FASTEVAL does an initial pass through the buffer and replaces all the opcodes by the address of the corresponding function that EVAL would have called. This expands the buffer 16-fold, but the expanded buffer is only used during evaluation and can be reused by every member of the population. Thus, originally, EVAL processed the tree $T + 1$ times (for T=48 test cases).

The Intel AVX instructions process up to 16 floating point data simultaneously. The AVX version of EVAL was rewritten to take advantage of this. Indeed as we expect trees that are far bigger than the CPU cache ($\approx$16 million bytes, depending on model), EVAL was rewritten to process each tree's buffer only once. This is achieved by EVAL processing all of the test cases for each opcode, instead of processing the whole of the tree on one test case before moving on to the next test case. Whereas before each recursive call to EVAL returned a single floating point value, now it has to return 48 floating point values. This was side stepped by requiring EVAL to maintain an external stack where each stack level contains 48 floating point values. The AVX instructions operate directly on the top of this stack and EVAL keeps track of which instruction is being interpreted, where the top of the stack is, and (with PTHREADS) which thread is running it. Small additional arrays are used to allow fast translation from opcode to address of eval function, and constant values[7]. AVX instructions are used to speed loading each constant into the top stack frame. Similarly all 48 test cases (x) are rapidly loaded on to the top of the stack. However, the true power of the implementation comes from being able to use AVX instructions to process the top of the stack and the adjacent stack frame (holding a total of 96 floats) in essentially three instructions to give 48 floating point results.

The depth of the evaluation stack is simply the depth of the GP tree. GPquick uses a fixed buffer length for every individual in the GP population. This is fixed by the user at the start of the GP run. Fixing the buffer size also sets the maximum tree size. Although in principle this only places a very weak limit on GP tree depth, it has been repeatedly observed (Langdon, 2000b) that evolved trees are roughly shaped like random trees. The mathematics of trees is well studied (Sedgewick and Flajolet, 1996), in particular the depth of large random binary trees tends to a limit $2\sqrt{\pi \lceil \text{treesize}/2 \rceil}$ + O(tree size$^{1/4+\epsilon}$) (Sedgewick and Flajolet, 1996, page 256). (See Flajolet limit in Figures 10, 11, 16, 17, 18 and 26.) Thus the user-specified tree size limit can

[7] In (Langdon, 2020a) evolution was used to investigate other opcode translation speed ups.

be readily converted into an expected maximum depth of evolved trees. The size of the AVX eval stack is set to this plus a suitable allowance for random fluctuations and O(tree size$^{1/4+\epsilon}$). Note, with very large trees, even allowing for the number of test cases and storing floats on the stack rather than byte-sized opcodes, the evaluation stack is considerably smaller than the genome of the tree whose fitness it is calculating.

Although AVX allows reduction operations across a stack frame, these are not needed until the final conversion from output to fitness value. However although faster, the reduction operations manipulate the 48 numbers in a different order and so may (within floating point tolerances) produce different answers. Since the reduction is a tiny part of the whole fitness evaluation we decided instead to ensure the AVX version produces identical results to the original system and so the final fitness evaluation is done with a conventional for loop.

2.3 PTHREADS GPquick

The second major change to GPquick was to delay fitness evaluation so that the whole new population can have its fitness evaluated in parallel. (This means PTHREADS can only be applied when GPquick is operating in generational mode.) If pThreads > 0, the population fitness evaluation is spread across pThreads. As trees are of different sizes, each fitness evaluation will require a different time. Therefore which tree is evaluated by which thread is decided dynamically. Due to timing variations, even in an otherwise identical run, which tree is evaluated by which thread may be different. However great care is taken, including considerable regression testing, so that this cannot affect the course of evolution. E.g., everyone in the population has their correct fitness no matter what order they were evaluated and pseudorandom numbers are only generated in sequential code. (Although, for example, where trees have identical fitness, it can affect which is found first and therefore which is reported to the user).

EVAL requires a few data arrays. These are all allocated at the start of the GP run. Those that are read-only can be shared by the threads. Each thread requires its own instance of read-write data. To avoid "false sharing", care is taken to align read-write data on cache line boundaries (64 bytes), e.g. with additional padding bytes and `((aligned))`. This ensures each thread writes to its own cache lines and therefore these cached data are not shared with other threads.

Initially surprisingly an almost doubling of speed was obtained by also moving crossover operations to these parallel threads. Since crossover involves random choices of parents and subtrees these were unchanged but instead of performing the crossover immediately a small amount of additional information was retained and to be read later by the threads. This allows the crossover to be delayed and performed in one of up to 96 C++ pthreads. The results are identical but give an additional ≈two-fold speed-up.

One gotcha we found during development of the multi-threaded code is that (in at least some versions of the GNU C++ runtime library), although heap management `new` and `delete` operations are supported in threads, we experienced considerable slow-down when allocating and deleting large buffers within the multi-threaded code. In the newer code, used with runs using the middle-sized populations (500), and described in Sections 2.4 to 2.8, large buffers are created at the start of the run and reused. Similarly, all other (smaller) uses of the dynamic memory heap is done only in sequential, non-threaded, code.

2.4 Reducing Memory

As previously mentioned, in Section 2.1, GPquick had long ago been updated to allow either steady state populations or evolution via separate generations and that the separate generations mode effectively doubled the memory required (from M to $2M$ buffers). It was also known that (Koza et al., 1999) had described an implementation in which a generational GA uses the same memory as a steady state one[8]. (Koza et al., 1999) assumes a single computer. In (Langdon, 2020c) we described a simplified scheme for use with t threads and which, for 2 parent crossover, needs up to $M + 2t$ buffers. Using the tricks described in the following Sections (2.5–2.8) we were able to reduce this to about $(1 - e^{-1})M + t$. I.e., a generational GA using less memory than a traditional steady state GA.

Although details are given in (Langdon, 2020c), we next give the gist of our (multi-threaded) implementation of (Koza et al., 1999)'s idea. Before the new generation is created, the old population is sorted into two queues according to the number of children they will have in the next generation. (We optimise (Koza et al., 1999) by immediately deleting those that have no children.) The first queue contains parents with exactly one child. Unlike (Koza et al., 1999), our 2 parent crossover creates only one child at a time, and therefore we need only two queues. The second queue contains all the parents with two or more children. The queues contain all the information about the crossovers: in particular which child is to be created and who the other parent is.

The new generation is created in the order of the parents (not the order of the children). A parent is selected from the first queue. If that is empty at any point, a parent is taken from the second queue. The child is created using that parent and its mate (which may be on either queue). The number of children belonging to both parents is decremented. When the child count goes to zero on either parent, we are done with that parent. It can be deleted and its buffer used by the next generation. If the child count is now one on either parent, that parent is transferred from the second queue to the first queue. (Note self-crossover is allowed, so a parent's child count maybe reduced by two as well as by one. Thus a parent can be removed from the second queue and deleted, without passing through the first queue). This continues until both queues are empty, at which point the new generation will have been created and the old one will have been deleted.

On a multi-core computer (with t threads), to allow crossover to create children in parallel, rather than 2 spare buffers, we need (up to) $2t$ spare buffers. Locks are used to safely allow multiple threads to update the two queues and to keep track of the buffers used to store the trees.

Sections 2.6–2.8 describe further refinements to reduce both crossover overhead and actual memory usage.

2.5 Incremental Evaluation

The implementation of incremental evaluation is fully described in (Langdon, 2021b). Briefly we exploit the fact none of the components of the evolved trees have side effects. (They are pure functions.) And thus the GP trees can be evaluated in any order.

[8]During 2 parent crossover both steady state GAs and (Koza et al., 1999) need 2 spare buffers. So the number of memory buffers required is $M + 2$.

In particular they can be evaluated from the crossover point towards the root node, see Figures 4 and 5. The evaluation on reaching the root node is identical to the conventional top-down recursive evaluation starting at the root node.

In converged populations, not only does most of the population have the same fitness but many children have the same evaluation as their parents. By evaluation from the change point (rather than the root node), it is easy to spot when the child's evaluation is the same as its parent, Figure 5. It turns out that enormous savings can be made because, even in simple floating point operations, the runtime impact of, even large, changes is quickly dissipated (Langdon et al., 2021a) and so only a tiny fraction of the child has to be evaluated before proving that its fitness is the same as that of its parent, see Figure 21. Figure 23 shows that on average the number of steps up the tree before the parent and the child evaluations synchronise, and thus the crossover change becomes invisible, is remarkably small compared to the distance from the change to the root node. We will discuss the implications for complex systems of the resistance to change by large programs in Section 6.

Having considerably sped up fitness evaluation, in these huge trees, the cost of crossover creating the children becomes dominant. The next three sections describe innovations to reduce its cost, without changing the path of evolution, in a multi-threaded generational evolutionary algorithm.

2.6 Fatherless Crossover

For ease we will refer to the two parents as mum and dad (father). Subtree crossover produces one child tree at a time (Figure 6). The parent which donates its root node to the child is called the mum, and the one which donates a subtree is the dad. As trees become very large, the size of the inserted (dad's) subtrees remains much the same size, so the vast majority of the child's opcodes come from the mum.

The trick in fatherless crossover is simply to copy all the subtrees to be donated in the next generation before it starts.

As mentioned in Section 2.4, before starting the next generation, the number of children each tree will have is calculated. Each time a father subtree is copied, its child count is decremented, effectively copying the subtree has started the process of creating its child. If a tree's child count reaches zero, its buffer can be freed.

We call this "fatherless" crossover since the dad tree takes no further part in the crossover, and in many cases it can be deleted immediately.

In a multi-threaded generational GA (with t threads) using crossover up to $M + 2t$ buffers are needed to store the population (size M) whilst it is created by crossover (Section 2.3, (Langdon, 2020b)). Since the donated subtrees are so small compared to the trees they are drawn from, they can be neglected, therefore with fatherless crossover only up to $M + t$ buffers are needed.

In converged populations, there is no fitness differential, and so the allocation of children to parents becomes random. In large converged populations, inplace crossover, fitness first (following sections) and fatherless crossover together reduce memory consumption to about $(1 - e^{-1})M + t$ (Langdon, 2021a, sec. 8).

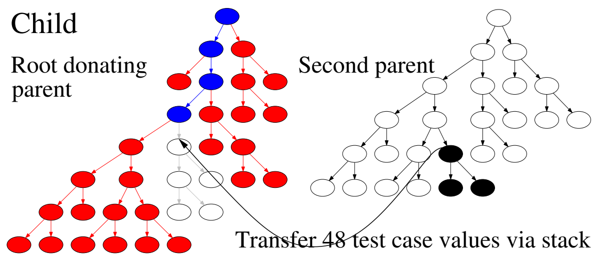

Figure 4: Incremental evaluation. The subtree to be inserted (black) is evaluated on all test cases and the values are transferred to the evaluation of child (left) at the location of the subtree to be removed (white). Differences between original code (white subtree) and new are propagated upwards (blue nodes) until either all differences are zero or we reach the root node.

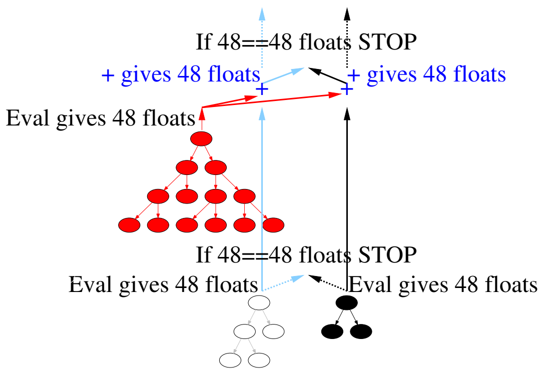

Figure 5: Incremental evaluation begins by recursively evaluating the subtree to be removed from the mum (white) and the subtree to be inserted (black). If they are the same, evaluation STOPs, the child's evaluation is the same as the mum's and so its fitness must be the same. If not, evaluation does the operation above the crossover point. To do this it must find the evaluation of the other side tree (red). This is found by recursively evaluating it. Note the side subtree is the same in both parent and child and so need be evaluated only once per test case. If evaluation in the mum and child are now identical the evaluation STOPs. Otherwise evaluation proceeds up the mum's tree until either the evaluation in the mum and (unborn) child are the same or it reaches the root node. (Cf. blue nodes in Figure 4.)

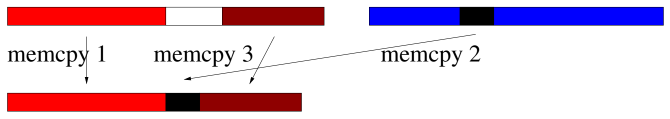

Figure 6: GPquick (Singleton, 1994) crossover. In GPquick GP trees are flattened. Effectively the tree is stored by recursive depth first traversal and each new tree node's opcode is appended to a linear buffer. Thus subtrees are contiguous bytes in the linear buffer. The maximum tree size is specified by the user before the GP is started. To avoid heap fragmentation all tree buffers are this maximum size. GPquick subtree crossover requires three memcpy buffer copies: 1) root segment of donating parent (mum, red/brown) is copied to offspring buffer. 2) subtree from second parent (dad, blue/black) is copied to offspring. 3) tail (brown) of 1st parent copied to child.

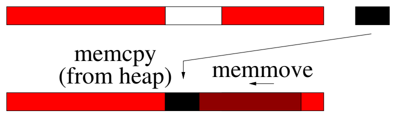

Figure 7: Inplace subtree crossover can only be used if the offspring is the mum's last child as it reuses her buffer. 1) Copy of dad subtree overwrites mum's buffer. 2) In 71% of children the subtree to be removed (white) and to be inserted (black) are different sizes, and so memmove is used to shuffle the second part of mum's buffer (brown) up or down.

2.7 Inplace Crossover

In order to minimise memory consumption, Section 2.4, a count of how many children each parent has is kept. As each child is created, this count is decremented. When the count is one, crossover is about to create the last child for a particular mum.

In the case of the last child, the mum's buffer can be immediately re-used for the child. On average about half the child is identical to its mum and so re-using the buffer immediately avoids about half crossover's work, Figure 7.

Also it turns out the shuffle operation, memmove, (needed to repack the child's buffer, when the removed and inserted subtrees are not the same size) is actually faster than memcpy (Langdon, 2021a, Sec. 5). Note Figure 7 also shows Fatherless crossover, in that it shows the inserted crossover fragment, black, being taken from the heap, rather than copied from the dad tree (as in the original GPquick crossover, Figure 6).

2.8 Evaluating Fitness Before Crossover

With diverse populations most of the children are allocated to the fittest parents. This means most of the population do not have children. Apart from SUS (Baker, 1987), with commonly used stochastic selection schemes (e.g. tournament selection) parents are chosen independently. Thus in large uniform populations with 2 parent crossover, on average e^{-2} of the population do not have children. With mutation only or fatherless crossover, Section 2.6, this rises to e^{-1} of the population. Notice individuals in a GA population which do not have children, have no impact on the course of evolution. Since rapid fitness evaluation means creating large trees becomes the dominant computational cost, speedups of about $e^{-1} = 37\%$ can be obtained by not creating them.

Table 1: Long term 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 input). 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:	Panmictic, non-elitist, generational.
Parameters:	Initial population (4000) ramped half and half Koza (1992) depth between 2 and 6. 100% unbiased subtree crossover. 100 000 generations (stop run if any tree reaches limit 15×10^6).

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

To decide how many children will be allocated to each tree, we reverse the usual order of crossover and fitness operations, and use incremental fitness evaluation to evaluate each child in the next population before creating them. After fitness selection we know who in the next generation will have children and who will not. Those without children need not be created.

To evaluate fitness before crossover only a small change is needed to incremental evaluation (Section 2.5). All we need do is evaluate using the mum's tree rather than that of the (as yet unborn) child. Referring to Figure 4, the unchanged code (red and blue nodes) is obviously identical in the mum and the child and so evaluation can use the mum's buffer and produce identical results to those that would be obtained using the child's buffer, if we waited for crossover to create the child in the usual way.

3 Experiments

We use the well known Sextic polynomial benchmark (Koza, 1994, Tab. 5.1). Briefly, the task given to GP is to find an approximation to a sixth order polynomial, $x^6 - 2x^4 + x^2$, given only a fixed set of samples, i.e., a fixed number of test cases. For each test input x we know the anticipated output $f(x)$, see Figure 8 and Table 1. Of course the real point is to investigate how GP works and how GP populations evolve over time. We ask ourselves whether it is possible for GP to continue to find improvements, even for such a simple continuous problem, as Lenski's *E. coli* experiments are showing, or, like the Boolean case (Langdon, 2017), whether the GP population will get stuck early on and from then on never make further progress. Note that we here make use of crossover exclusively, so no random mutations are allowed to introduce any new genetic material during the run. All the variation the algorithm can make use of must be present in the first generation.

We ran three sets of experiments. In the first the new GP system was set up like the original Sextic polynomial runs which reported phenotypic convergence (Langdon et al., 1999, Fig. 8.5). The first set uses a population of 4000, the second 500 and the last 48.

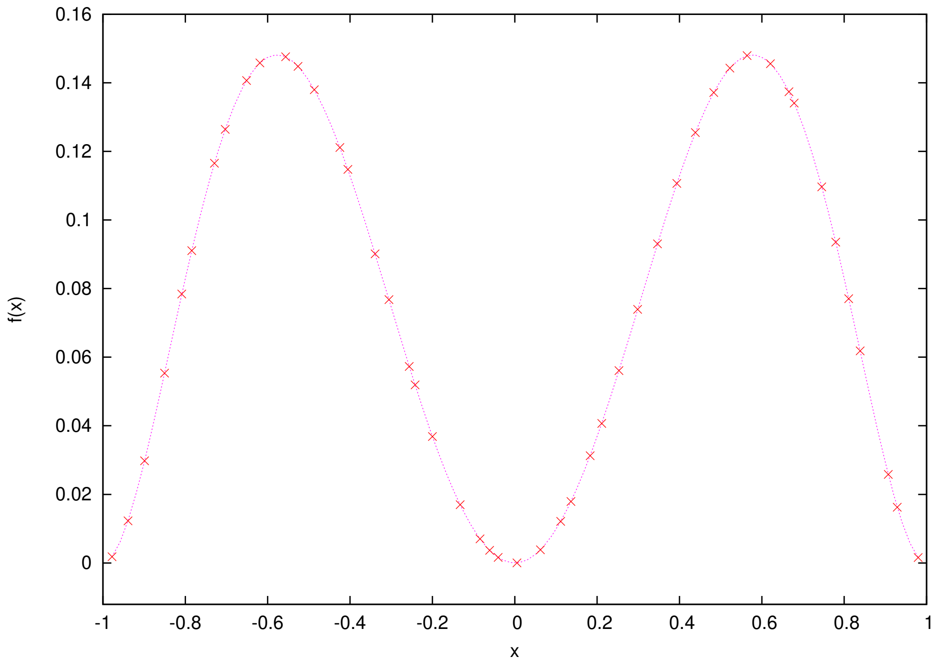

Figure 8: 48 test cases for Sextic Polynomial Benchmark $x^2(x-1)^2(x+1)^2$.

3.1 Crossover

Each generation is created entirely using Koza's two parent subtree crossover (Koza, 1992). (GPquick creates one offspring per crossover.) For simplicity and in the hope that this would make GP populations easier to analyse, both subtrees, the one to be removed and the one to be inserted are chosen uniformly at random. That is, we do not use Koza's bias in favour of internal nodes (functions) at the expense of external nodes (leafs or inputs). Instead, the root node of the subtree (to be deleted or to be copied) is chosen uniformly at random from the whole of the parent tree. This means there is more chance of subtree crossover simply moving leaf nodes and so many children will differ from the root node donating parent by just one leaf.

As mentioned above, once fitness evaluation has been sped up by parallel processing, for very big trees producing the child is a surprisingly large part of the remaining run time and so it, too, can be implemented in parallel. However, the choice of crossover points is done in sequential code and remains unaffected by multithreading. This ensures the variability introduced by multiple parallel threads does not change the course of evolution.

3.2 Fitness Function

The fitness of every member of every generation is calculated using the same fitness function as (Koza, 1994, Tab. 5.1). That is, barring rounding errors (Section 2.1), fitness is given by the mean of the absolute difference between the value returned by the GP tree on each test case and the Sextic polynomial's value for the same test input (see Table 1). We use tournament selection to choose both parents.

Like (Koza, 1994, Tab. 5.1), we also keep track of the number of test cases where each tree is close to the target (i.e. within 0.01, known as a "hit"). The number of hits is used for reporting the success of a GP run. It is not used internally during a GP run. Also, our GP runs do not stop when a solution is found (48 hits) but continue until either the user-specified number of generations is reached or bloat means the GP runs out of memory.

Where needed, floating point calculations are done in a fixed order, to avoid parallelism creating minor changes in calculated fitness, which could quickly cause otherwise identical runs to diverge because of implementation differences in parallel calculations. (Also mentioned above in Section 2.2.)

4 Results

4.1 Results Population 4000 trees

In the first set of experiments, we use the standard population of 4000 trees. Table 2 summarises the results of 10 runs. In all cases GP found a reasonable approximation to the target (the Sextic polynomial). Indeed in all but one run (47 hits) the best trees score 48 out of 48 possible hits. I.e. they are within 0.01 on all 48 test cases. Indeed in most cases the average error was less than 10^{-4}. Figure 1 shows that GP tends to creep up on the best match to the training data. Typically after several thousand generations, GP has progressively improved by more than a thousand increasingly small steps. (See Table 2 column 3 and Figure 1).

In all ten runs with a population of 4000, we see enormous increases in size and all are stopped as they hit the size limit (15 000 000) before reaching 100 000 generations. Column 5 in Table 2 gives the size (in millions) of the largest evolved tree in each run. The log-log plot in Figure 9 shows a typical pattern of subquadratic (Langdon, 2000a) increase in tree size. The straight line shows a power law fit. In this run the best fit has an exponent of 1.2. Column 6 of Table 2 shows that the best fit between generations ten and a thousand for all 10 runs varies between 1.1 and 1.9.

As expected not only do programs evolve to be bigger but also they increase in depth. As described above (Section 2.2), highly evolved trees tend to be randomly shaped and so as expected tend to lie near the Flajolet limit, depth $\approx \sqrt{2\pi|\text{size}|}$ (see Figures 10 and 11). (This is also true in the pop=500 and pop=48 runs, see following sections.)

Figure 12 shows that over the ten runs, most fitness improvements occur in trees of depth between 231 and 445, whilst most trees, regardless of fitness, are between depth 347 and 2345. That is, as the runs progress and deeper trees evolve the rate of innovation falls but nonetheless Figure 1 shows we still see fitness improvements later in the runs.

In all ten runs we see some phenotypic convergence. The "conv" column in Table 2 shows the peak fitness convergence. That is, out of 4000, the number of trees having exactly the same fitness as the best in the population. Typically at the start of the run (see Figure 13), the population contains mostly trees with poorer fitness, but later in the run the population begins to converge and towards the end of the run we may see hundreds of generations where more than 90% of the population have identical fitness.

Table 2: 10 Sextic polynomial runs with population 4000 (Section 4.1). gens^i is the power law fit of tree size between generation 10 and 1000. conv is the maximum number of trees in the population with identical fitness. The last column is the GP (before enhancements in Sections 2.4–2.8) speed on 48 core 2.60GHz Intel Xeon Gold 6126 CPU. Runs took between 2:18 and 15:15 hours (median just under 5 hours).

Gens	error $\times 10^{-9}$	impr[9]	hits	size $\times 10^6$	gens^i	conv	GPop/sec $\times 10^9$
6370	64487	2139	48	14.329	1.200	3981	58.2
8298	145796	2040	48	14.102	1.916	3982	57.4
2323	642006	389	47	13.441	1.387	3995	51.6
7119	507600	608	48	13.668	1.589	3997	55.0
11750	1	3583	48	13.854	1.364	3989	49.8
3412	65561	1277	48	14.348	1.625	3986	45.4
5106	71288	1615	48	14.233	1.146	3988	53.6
6112	728757	1871	48	14.500	1.254	3983	52.9
6679	28853	1741	48	14.022	1.396	3998	43.4
4454	67817	790	48	14.900	1.227	3997	54.9

[9]Figure 1 gives number of generations which improve on their parents, whereas here we give strictly better than anything previously evolved. Hence slight differences.

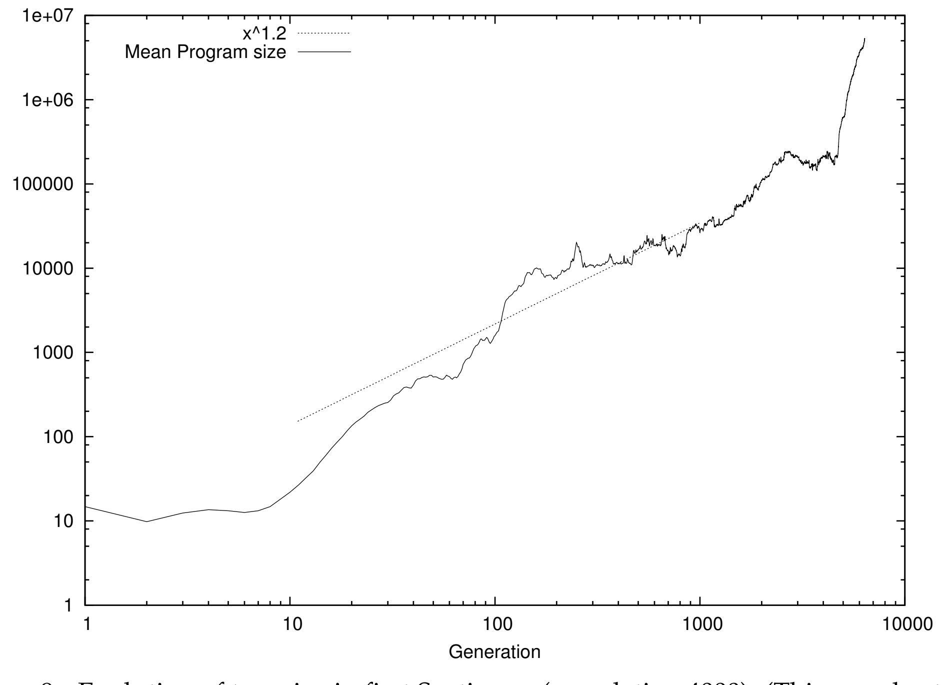

Figure 9: Evolution of tree size in first Sextic run (population 4000). (This run aborted after 6370 generations by first crossover to hit 15 million node limit.) Straight line shows best RMS error power law fit between generation 10 and 1000, $y = 8.65x^{1.2001}$.

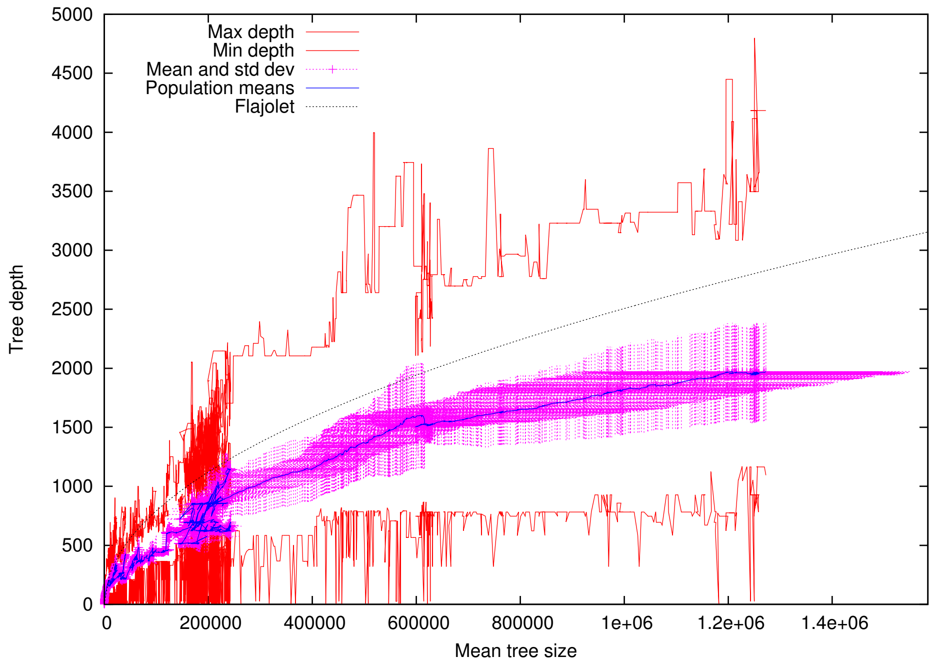

Figure 10: Evolution of depth and size in population of 4000 trees in typical Sextic polynomial run. In this run, as predicted (Langdon, 1999b), average trees lie within one or two standard deviations of random binary trees (Flajolet limit, depth $\approx \sqrt{2\pi|\text{size}|}$, (Sedgewick and Flajolet, 1996), dotted parabola). See also Figure 11.

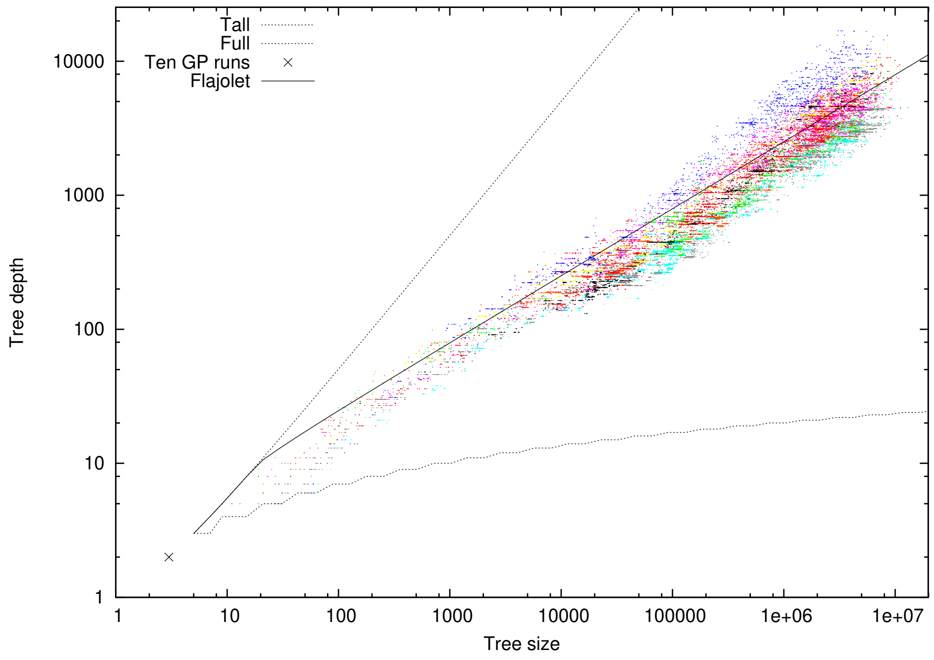

Figure 11: Plot of size and depth of the best individual in each generation for 10 Sextic polynomial runs with population of 4000. Binary trees must lie between short fat trees (lower curve "Full") and "Tall" stringy trees. Most trees are randomly shaped and lie near the Flajolet limit (depth $\approx \sqrt{2\pi|\text{size}|}$, solid line, note log-log scales). Figure 10 shows the first run in more detail.

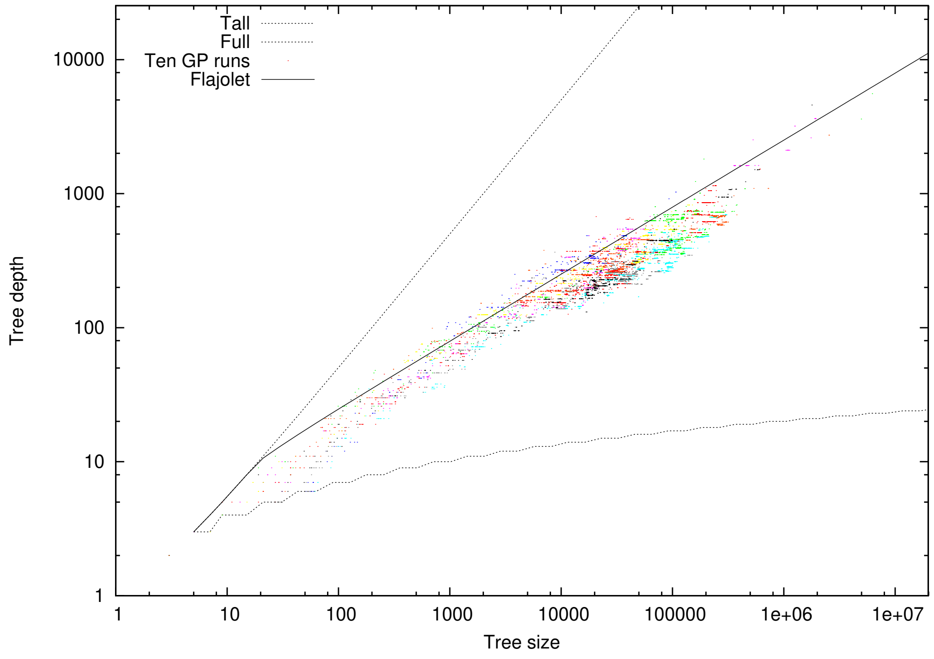

Figure 12: As Figure 11 but only show trees which are better than those earlier in each run.

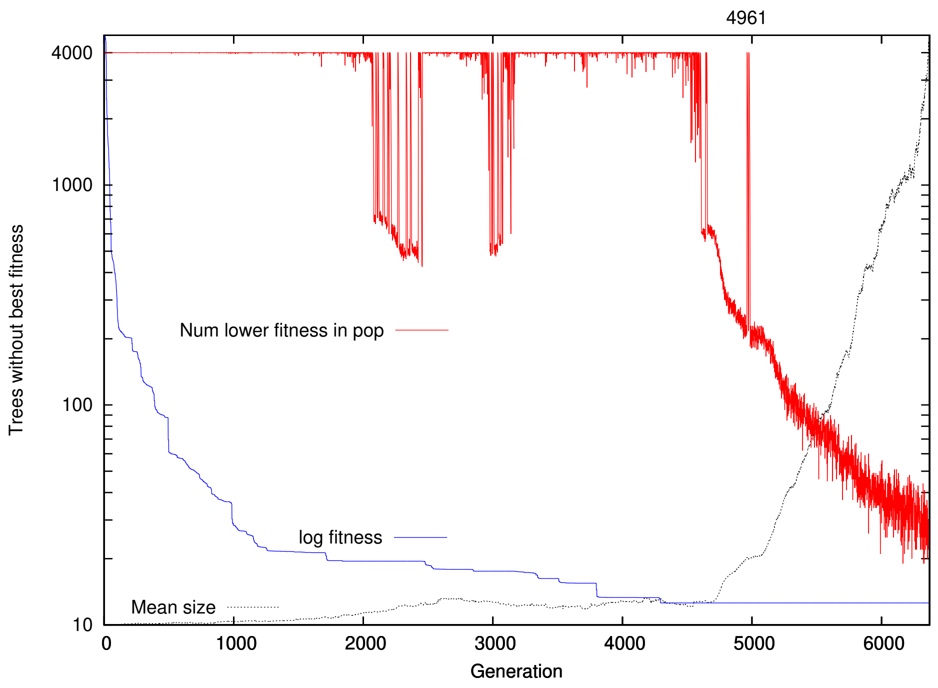

Figure 13: Fitness convergence in first Sextic polynomial pop=4000 run. Perhaps because of the continual discovery of better trees before generation 4975 and the larger population size, although the number of trees without the best fitness falls, unlike in the earlier Boolean problem (Langdon, 2017) and smaller populations, it never reaches zero. Notice tiny fitness improvement in generation 4961 resets the population for ten generations. (Mean prog size (linear scale, dotted black) and best fitness (log, blue) plotted in the background.)

Under these circumstances, even with a tournament size as high as 7, many tournaments include potential parents with identical fitness. These, and hence the parents of the next generation, are decided entirely randomly. However, even in the most converged population there are at least two individuals with worse fitness. (In Figure 13 it is at least 19.) As we saw with the Boolean populations (Langdon, 2017), even this small number can be enough to drive bloat (Langdon and Poli, 1997) (albeit at a lower rate).

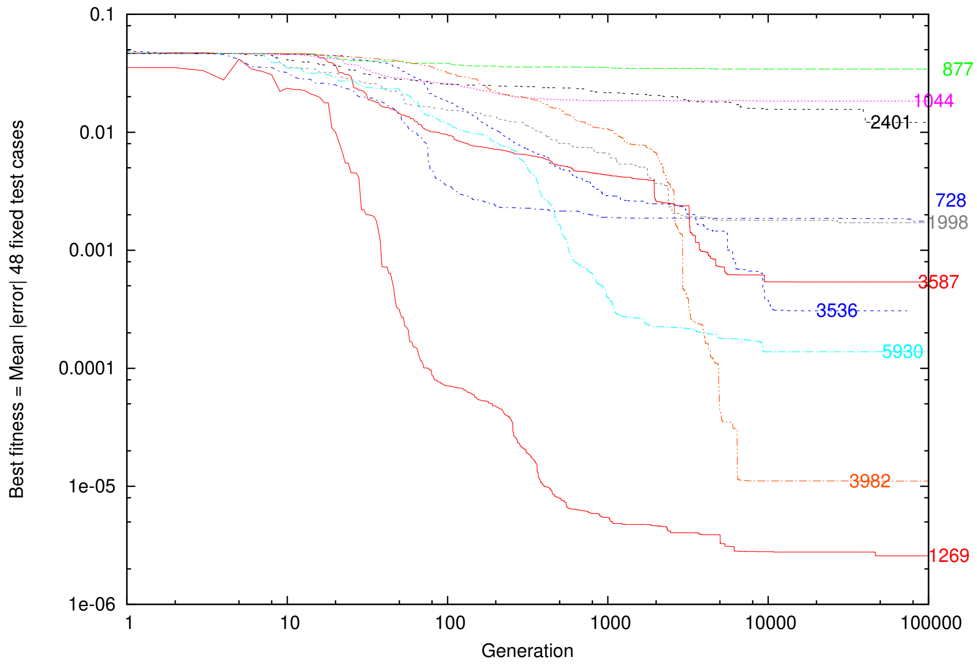

Figure 14: Evolution of mean absolute error in ten runs of Sextic polynomial (Koza, 1992) with population of 500. Runs to 100 000 generations (2 stopped early). Labels give number of generations when fitness got better.

4.2 Results Population 500 trees

We repeated the GP runs but allowed still larger trees to evolve by reducing the population from 4000 to 500 and splitting the available memory between fewer trees. Table 3 summarises these ten runs. Figure 14 shows the evolution of the best fitness with the reduced population size. Notice three runs do not really solve the problem and at best are on average more than 0.01 from the test cases. Nonetheless, in all cases evolution continues to make progress and each GP run finds several hundred or more small improvements (third column in Table 3).

Since we have deliberately extended the space available to GP trees, it is no surprise that the trees grow even bigger than before (column 4 in Table 3). Again bloat is approximately following a power law. Although in one unsuccessful run we see a power law exponent greater than 2, mostly growth is at a (sub-quadratic) rate similar to the bigger population runs (1.4–2.2 v 1.1–1.9, column 6 in Table 2 (pop 4000)). Figure 15 shows again as expected sub-quadratic growth in tree size between generations 10 and 1000. In fact the power law fit (<2.0), in the first GP run with a population of 500, seems to extrapolate well, even though the population starts to converge in later generations (see Figure 19 and also Figure 20).

Again randomly shaped trees evolve. Figure 16 shows the relationship between depth and size. As expected, in all runs the depth and size of the best trees in the population lie near the Flajolet limit, depth $\approx\sqrt{2\pi|\text{size}|}$, for large binary trees. Figure 17 shows the depth and size of the trees which are better than their parents and that the

Table 3: Ten Sextic polynomial runs with population 500 (Section 4.2). impr is the number of generations which are strictly better than anything previously evolved in the run. gens^i is the power law fit of tree size between generation 10 and 1000. The 7$^{\mathrm{th}}$ column is the newer GP's equivalent speed on a 3.00GHz Intel Xeon Gold 6136 server. (Due to scheduling constraints the runs used 16, 32 or 48 of the 96 available cores.) Runs took between 8 hours and 14 days (median 2 days 9 hours).

Gens	error $\times 10^{-6}$	impr	size $\times 10^{6}$	gens^i	conv	GPop/sec $\times 10^{9}$	threads
100000	538	3544	267.995	1.558	500	584.0	48
100000	34185	863	1046.200	1.736	500	748.6	48
73407	307	3484	2027.060	1.436	500	1103.4	16
100000	18349	1031	586.515	2.181	500	653.3	32
98876	137	5853	2040.590	1.928	500	639.7	32
100000	1765	664	98.936	1.408	500	400.0	32
100000	12116	2392	46.409	1.166	500	733.8	32
100000	11	3879	1093.950	1.351	500	666.0	16
100000	1709	1971	494.672	1.155	500	492.8	16
100000	2	1243	476.452	1.416	500	469.6	16

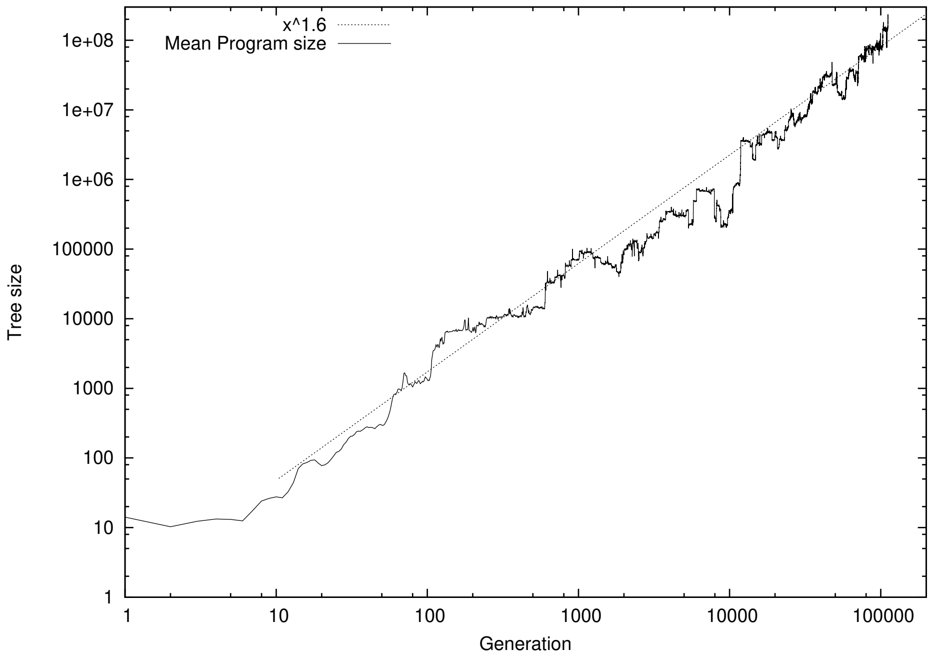

Figure 15: Evolution of tree size in first Sextic run (population 500). (This run aborted after 111 582 generations by first crossover to hit 400 million node limit.) Straight line shows best RMS error power law fit between generation 10 and 1000, $y = 1.3x^{1.56}$.

distribution of such innovators is also similar to that of random trees. Figure 18 shows the average tree similarly follows the best and the population as a whole also lies close to the Flajolet limit. (In the first few generations there is more variation in tree size than depth, so population mean sizes, Figure 18 (bottom left), appear too big for the "full line".

In contrast with Figure 16, Figure 17 shows only the points where evolution innovated. Most improvements occurred in trees of depth between 244 and 575. (Over the eight runs, most trees have depths between 3871 and 15 336.) That is, as with the larger population (Figures 11 and 12), there is more innovation in shallower trees. Nonetheless GP continues to find better programs, even though the population depth continues to grow, see Figure 14.

Unlike with the large populations, all the runs with populations of 500 trees showed some cases of complete fitness convergence ("conv" column in Table 3 is 500). Figure 19 shows, unlike with larger population (cf. Figure 13), in this run, the whole population has identical fitness 33 143 times (30% of the run). If we concentrate upon the last fitness improvement in generation 108 763 (2819 before the end of the run). This new improved Sextic polynomial performance takes over the whole population in half a dozen generations. (Shown in the plot (Figure 19) as the rightmost thin vertical red line.) However it fails to totally dominate the population in 861 (31%) of the remaining generations. Even though the mean number of lower fitness children is less than one (0.38) it is not zero, and this (given nearly three thousand generations) is still enough to double the average size of the trees. Figure 20 shows similar convergence with ten runs with a population of 500 but only up to generation 100 000. To avoid clutter, Figure 20 does not include size or fitness plots.

Figure 21 shows the speed-up given by incremental fitness evaluation (Section 2.5) in terms of the ratio of total opcodes to those actually evaluated. There is considerable variation between runs but the speed-up on average is about 60-fold. We should also caution that removing the fitness evaluation bottleneck means some other aspect of genetic programming (GP) now becomes the limiting factor. Figure 22 suggests that crossover bandwidth is the next hurdle to be overcome.

Figure 22 shows the speed (in terms of bandwidth) of the ten runs with a population of 500. Since these were run on a heavily used cluster, there is considerable noise. Nevertheless (see also last two columns in Table 3), there is no clear relationship between the number of computational cores used and speed. Of course the runs themselves are also highly variable (e.g. see also Figure 23). However Figure 22 suggests the GP is now, at least on the 3TB cluster node, memory bandwidth limited, rather than CPU bound.

Figure 23 plots the evolution of the mean number of nested function levels which totally conceal the crossover change when tested on the 48 test cases. To emphasise the stability of this average, although it is different for each run, rather than plotting against time, Figure 23 plots it against the average depth of the trees.

The mean (as a single statistic) does not show that there is considerable variation between members of the population. Occasionally a crossover will require the whole of the child to be evaluated, so all the functions between the crossover point and the root node must be evaluated. However *en bulk* successive generations are similar. Surprisingly, despite considerable evolution in the tree size and depth during the course

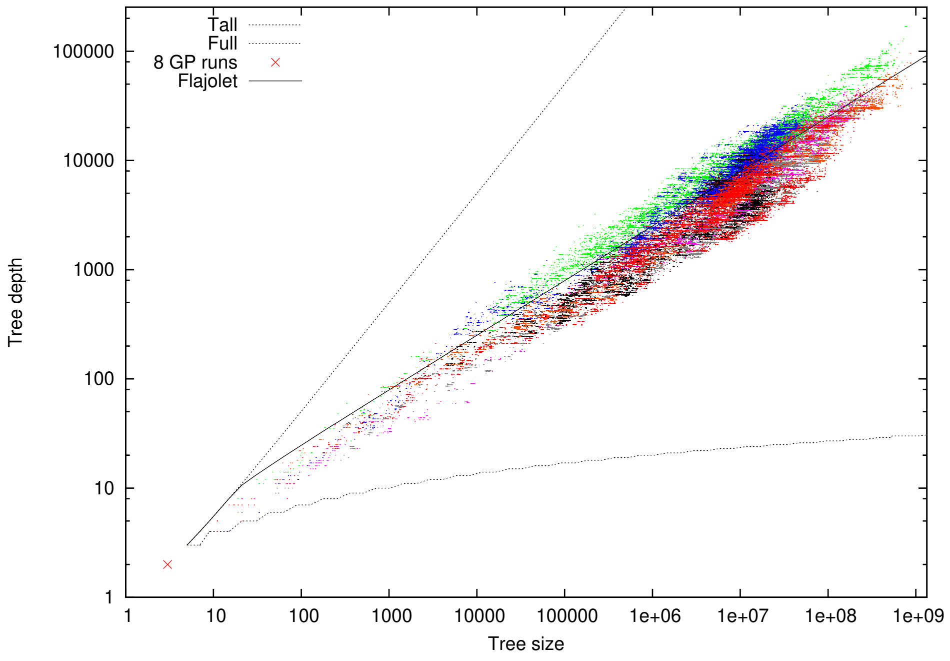

Figure 16: Plot of size and depth of the best individual in each generation for eight Sextic polynomial runs with population of 500 (depth data not collected in two runs). Binary trees must lie between short fat trees (lower curve "Full") and "Tall" stringy trees. Most trees are randomly shaped and lie near the Flajolet limit (depth $\approx\sqrt{2\pi|\text{size}|}$, solid line, note log-log scales). Same colours as Figure 14.

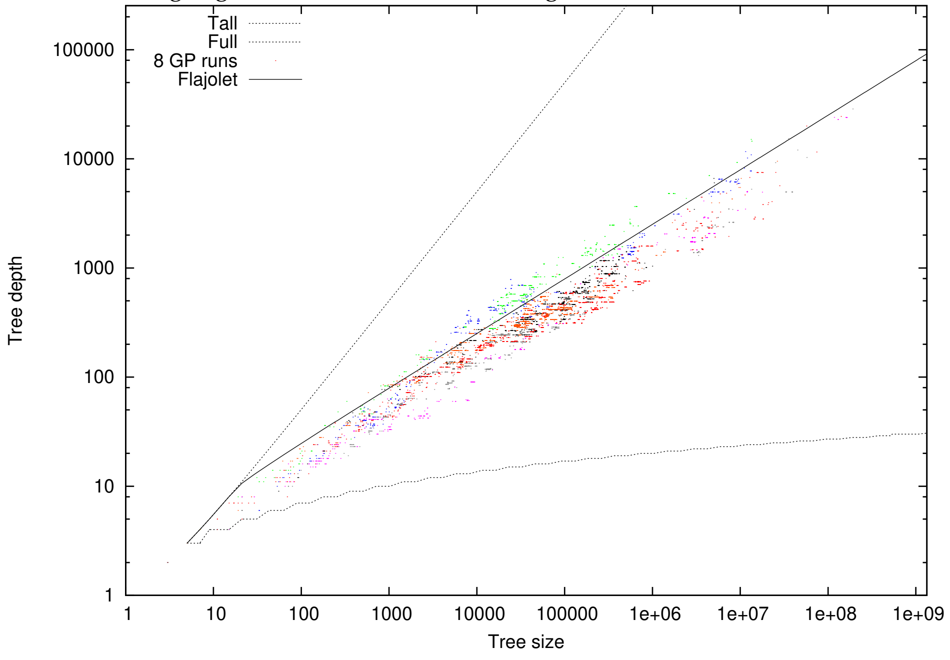

Figure 17: Plot of size and depth of improved individuals in eight Sextic polynomial runs with population of 500. As Figure 16 but only plot improvements.

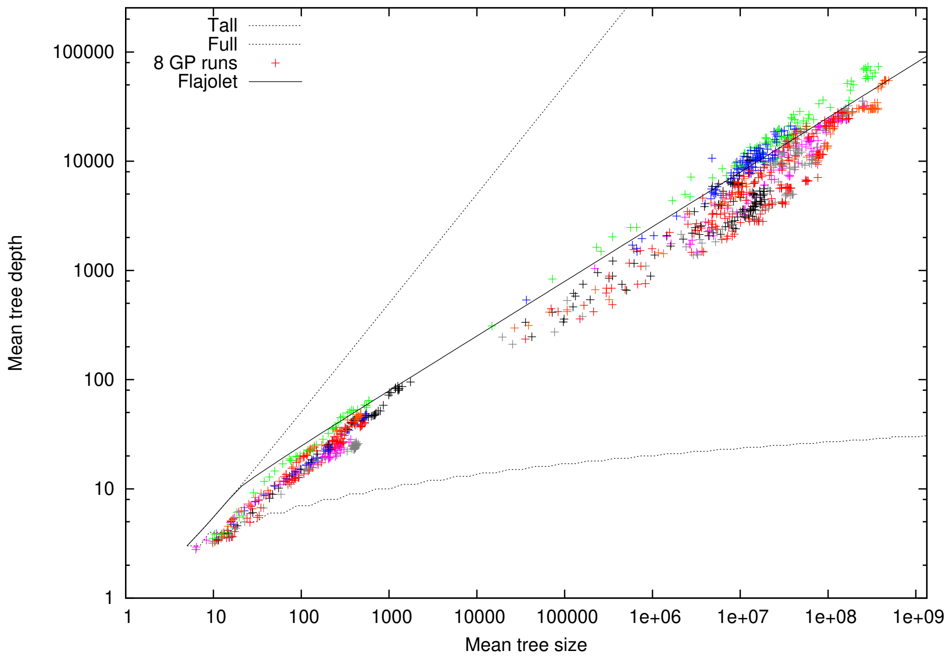

Figure 18: Plot of mean size and depth across population every 100 generations for eight Sextic polynomial runs with population of 500 (depth data not collected in two runs), Cf. Figure 16. On average trees are randomly shaped and lie near the Flajolet limit (depth $\approx\sqrt{2\pi|\mathrm{size}|}$, solid line, note log-log scales). Same colours as Figure 14.

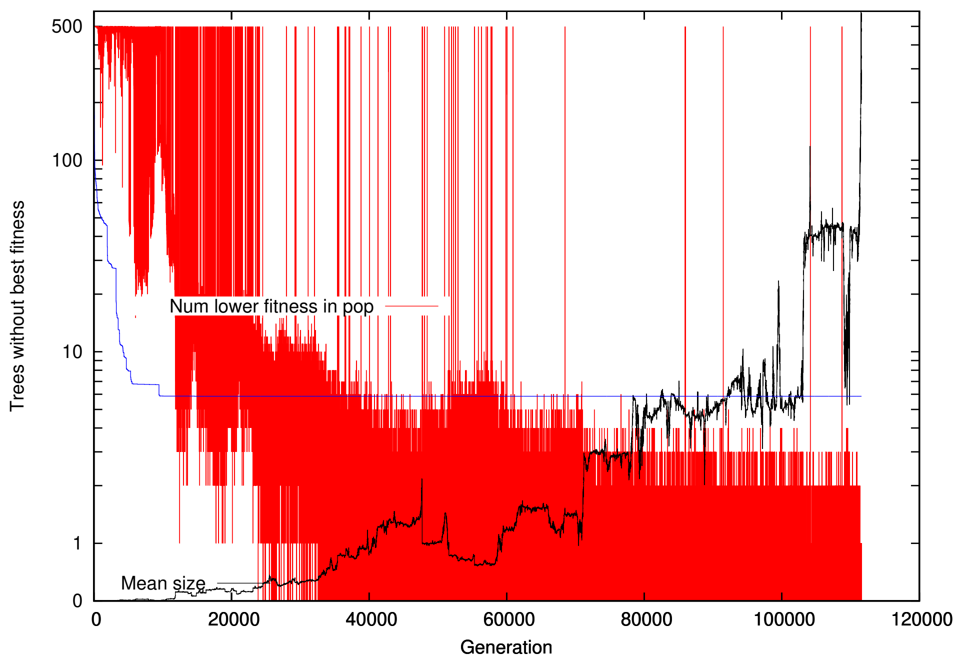

Figure 19: Fitness convergence in first Sextic polynomial pop=500 run. In 30% of this run, the whole population has identical fitness (y=0). Mean prog size (linear scale, black) and best fitness (log, blue) also plotted.

of 100 000 generations, the distribution of the number of upward moves in later generations is remarkably stable. This is reflected in the nearly horizontal lines for much of the plots. Each of the eight runs where depth data were collected, has evolved a different incremental fitness behaviour. Suggesting, although different between runs, the trees of a given run have evolved common structures (Langdon and Banzhaf, 2008), which are stable over tens of thousands of generations.

It appears that some test cases are better than others at detecting errors shrouded by intermediate computation. For example, 0.0 seems particularly poor at detecting differences. Also we would expect more tests to be more informative, however it appears that the effectiveness of test suites at detecting errors grows only slowly with the number of tests. In other work Langdon et al. (2021a) we show that often simple deeply nested continuous floating point functions are able to dissipate changes. That is, they do not have to be produced by genetic programming.

We have been primarily concerned with the impact on evolution, particularly the magnitude and rate of discovery of fitness improvements, and of information loss in highly nested functions. Nevertheless we will discuss its stifling of innovation, particularly in terms of evolvability and complex adaptive systems, in Section 6.

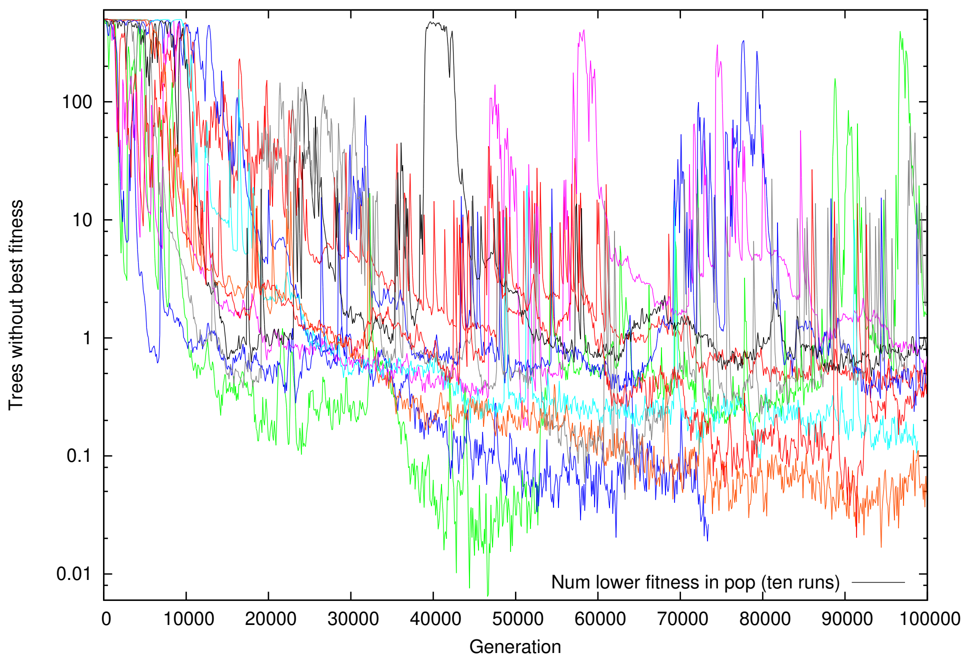

Figure 20: Fitness convergence in ten Sextic polynomial pop=500 runs, up to 100 000 generations. (Smoothed by plotting running average over 100 generations, cf. Figure 19.) Across the whole of each run and across the ten runs in almost half the generations the whole population has identical fitness. (The means for each individual run are between 25% and 63%.) Same colours as Figure 14.

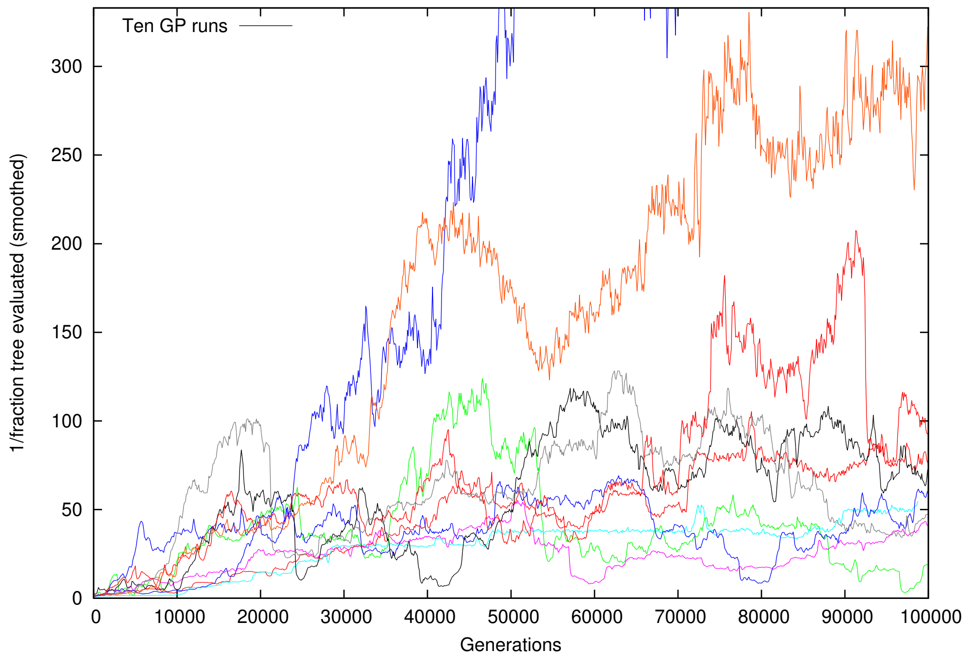

Figure 21: Impact of incremental evaluation (Section 2.5) on ten Sextic polynomial pop=500 runs. (Smoothed by plotting running average over 100 generations.) Fitness convergence and the huge trees later in the runs mean across the whole of each run on average only 1/60th of each tree need be evaluated to give its fitness exactly. (In ten runs: worst 1/30th cyan line, best 1/330th blue line.) Same colours as Figure 14.

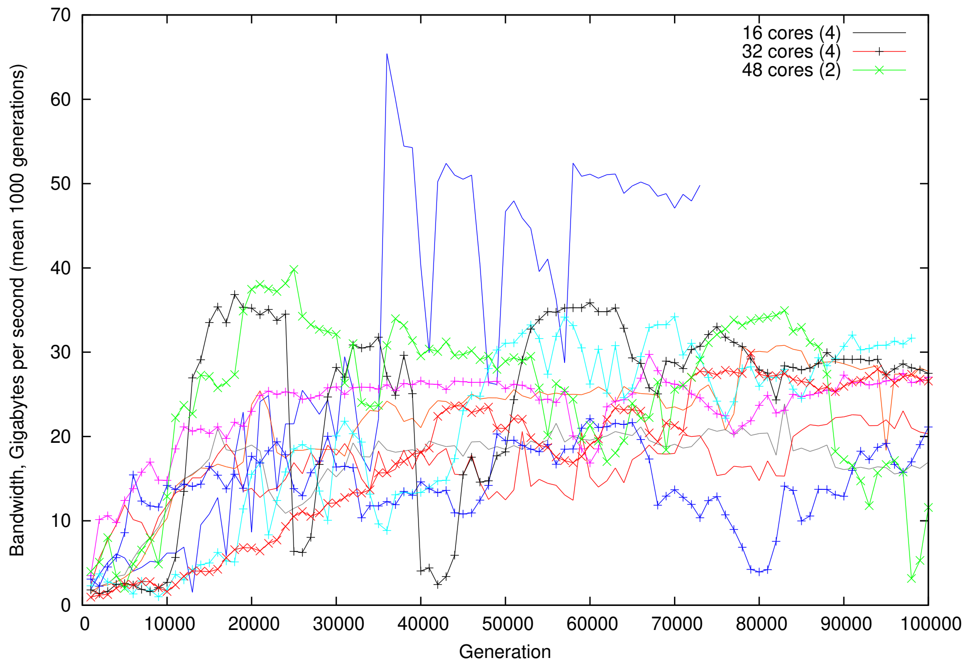

Figure 22: Bandwidth used by GP. To reduce noise inherent on multi-user servers, the mean over a 1000 generations is plotted. As is conventional in bandwidth calculations, each opcode (byte) of each GP tree is counted twice (once for reading and once for writing). The effective speed in terms of instructions per second is linearly proportional to bandwidth, since the number of test cases (48) is fixed. Same colours as Figure 14.

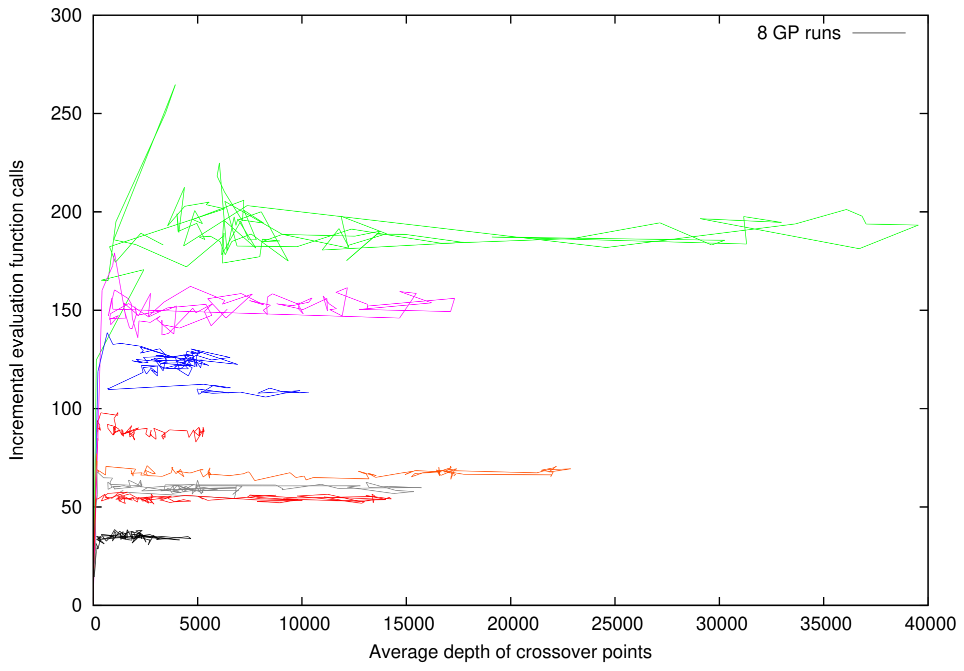

Figure 23: Mean number of moves up the tree taken by incremental evaluation (Section 2.5) on eight Sextic polynomial pop=500 runs. (Data collected every 1000 generations.) Same colours as Figure 14.

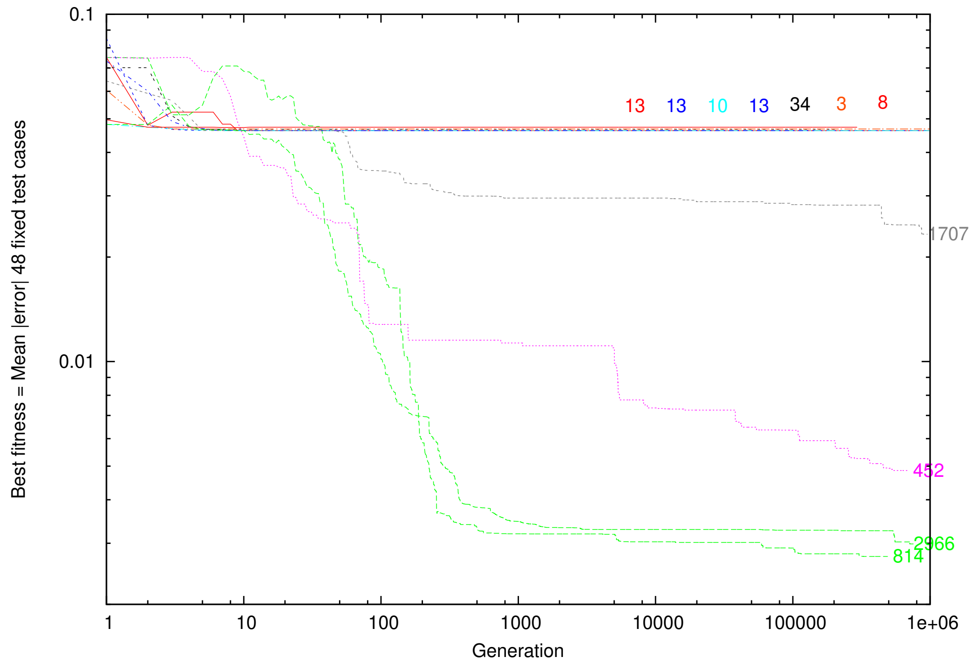

Figure 24: Evolution of mean absolute error in 11 runs of Sextic polynomial (Koza, 1992) with population of 48. (Up to a million generations or aborted on running out of memory, 500 million node limit.) End of run label gives number of generations when fitness got better. (Seven shown at top right to avoid crowding.)

4.3 Results Population 48 trees

In the final experiments the population was reduced still further to allow even larger trees to be evolved on the default 46 GB cluster server (Figure 2). These smallest population runs were run with a population of 48, since this should readily map to the available Intel multi-core servers.

With the small population, none of the runs solve the problem. Indeed only three runs got close on 40 or more test cases (see Figure 24 and Table 4). Of the remaining eight, only one finds a large number of fitness improvements. Seven runs have only between 3 and 30 generations with fitness improvements, column 3 in Table 4. Figures 3 and 25 show the enormous bloat in the first of these. In three of these, the population gets trapped at trees with just three nodes which evaluate to constants 0.0626506, 0.069169 and 0.0830508, although the population eventually escapes and large trees evolve by the end of the run. Except for these three runs, all the other runs contain populations where every member of the population has identical fitness. Therefore their maximum convergence is 48 (see “conv” column in Table 4). The final column is average speed, in giga GP operations/second.

As expected, as with larger populations, the highly evolved binary trees are again approximately the same shape as random trees. See Figure 26.

In contrast with Figure 26, Figure 27 shows only the points where evolution innovated. Figure 27 is dominated by the four runs which found most improvements (see

Table 4: 11 Sextic polynomial runs with population of 48 (Section 4.3). impr is the number of generations which are strictly better than anything previously evolved in the run. gens^i is the power law fit of tree size between generation 10 and 1000. conv is the maximum number of trees in the population with identical fitness. The last column is the GP (before enhancements in Sections 2.4–2.8) speed on 48 core 2.60GHz Intel Xeon Gold 6126 CPU. The runs took from 26 minutes to 8 days (median 39 hours).

Gens	error $\times 10^{-6}$	impr	hits	size $\times 10^6$	gens^i	conv	GPop/sec $\times 10^9$
1000000	46215	11	16	63.920	1.633	48	36.5
491618	2748	745	46	396.576	2.060	48	34.9
1000000	46215	7	13	190.654	1.448	48	57.4
689414	4857	448	40	159.949	1.260	48	38.1
1000000	46215	8	14	50.365	1.701	48	26.2
143251	46215	11	14	99.541	1.672	48	54.1
212528	46650	30	14	257.766	na	42	26.7
1000000	46730	3	14	0.000	na	42	.004
958147	23259	1683	18	308.958	1.791	48	53.5
294098	47174	3	12	308.121	na	43	24.5
757830	2985	2921	44	294.821	1.320	48	50.2

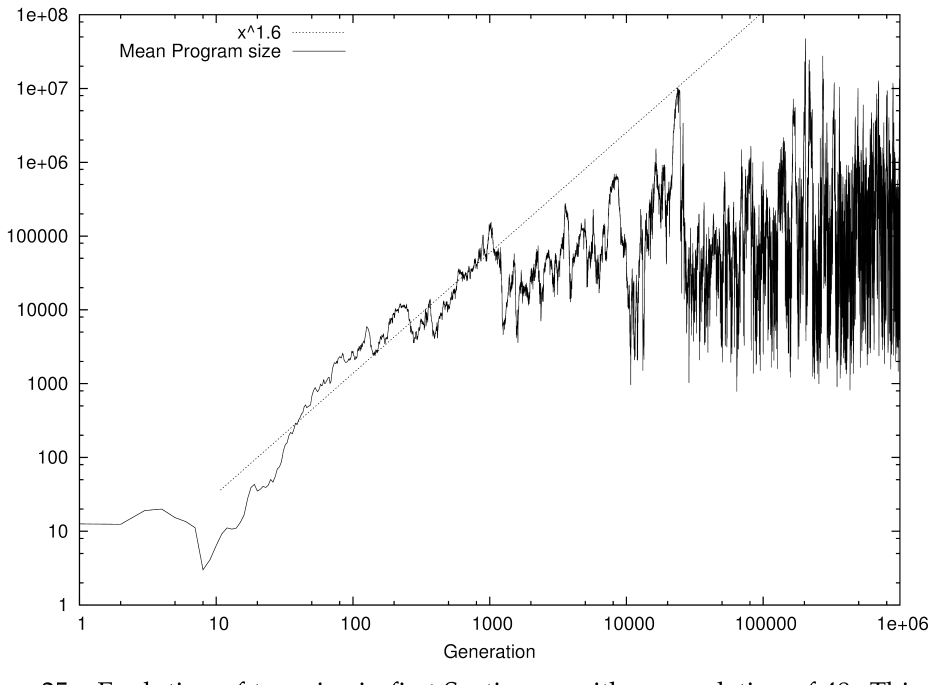

Figure 25: Evolution of tree size in first Sextic run with a population of 48. This run ran for a million generations. Straight line shows best RMS error power law fit between generation 10 and 1000, $y = 0.75x^{1.63}$.

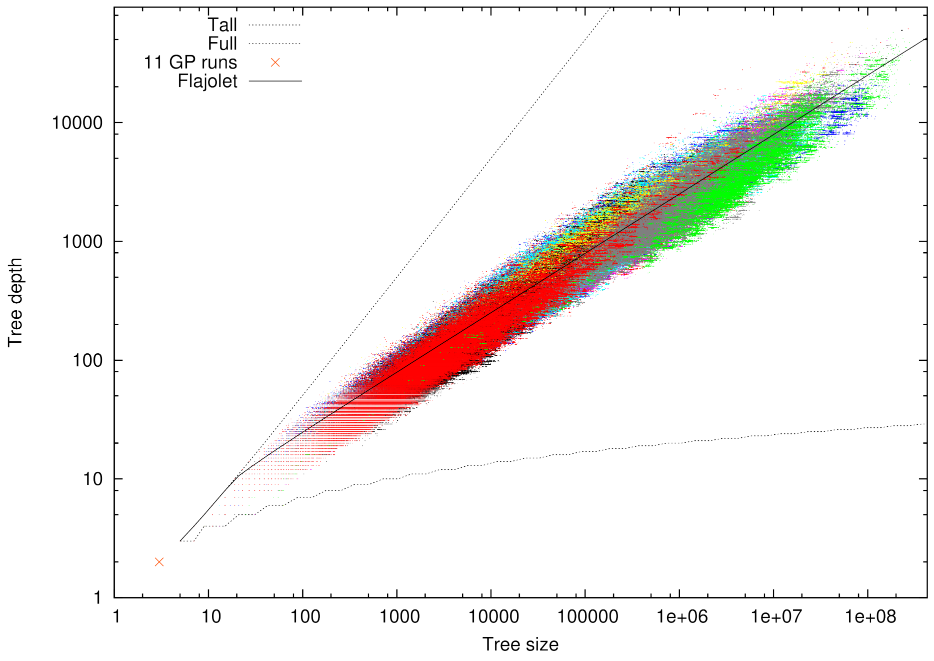

Figure 26: Plot of size and depth of the best individual in each generation for eleven Sextic polynomial runs with population of 48.

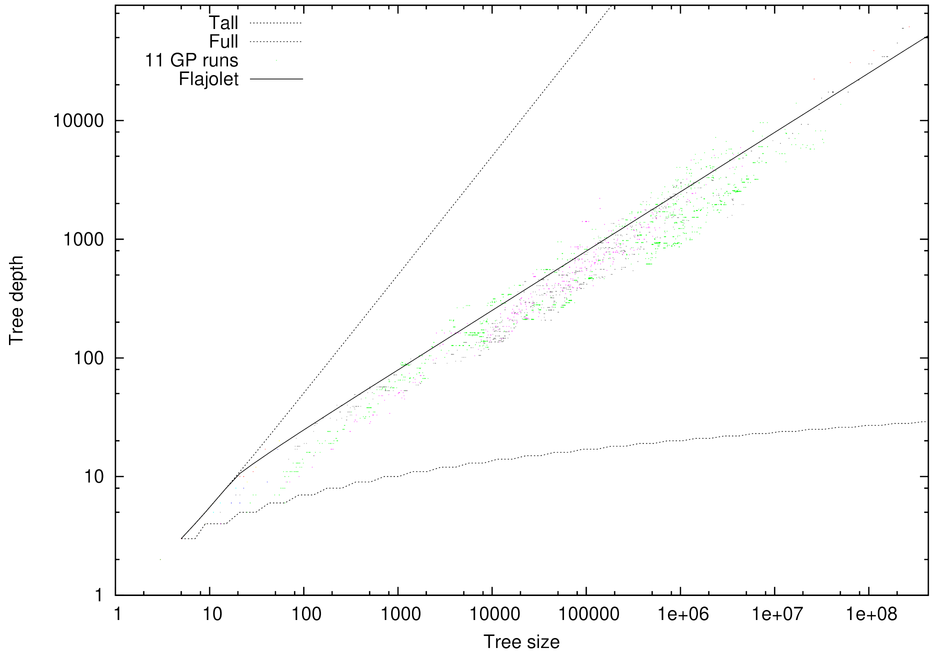

Figure 27: As Figure 26 but only plot discovery of new best in population fitness.

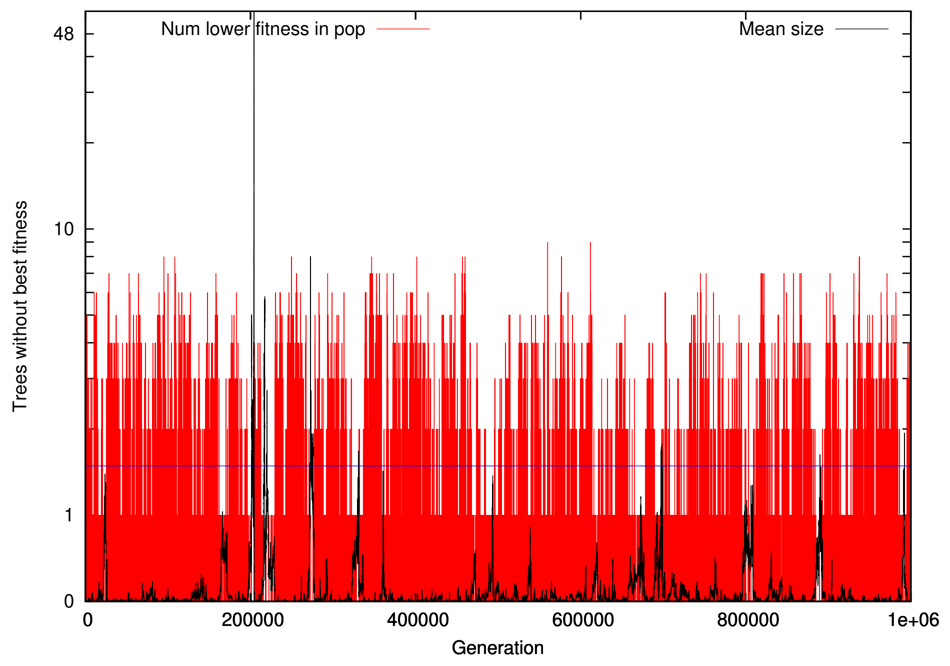

Figure 28: Fitness convergence in first Sextic polynomial run with population of 48 trees. After generation 19 the best tree in every generation has a fitness of 0.0462157 (4 hits) (tree returns 0.0769947 regardless of test case, cf. Figure 8). In 90% of this run, the whole population has identical fitness (y=0). (Mean number of poor fitness children is 0.1315.) Mean prog size (linear scale, black) and best fitness (log, blue).

Table 4 and plotted in black in Figure 2). Most improvements occurred in trees of depth between 211 and 1083 (median 465). (Over the eleven runs, most trees have depths between 337 and 4127, median 1401.) That is, as with the larger population (Figures 11 and 16), there is more innovation in shallower trees.

Figure 28 shows for almost the whole of the first run with 48 trees the best fitness in the population is fixed but once trees get big enough further size changes are essentially random (Figures 2 and 3). (Figure 29 shows similarly, albeit slightly less extreme, behaviour in the last population 48 run .) Notice fitness depends only on the sum of the absolute difference between the value returned by the GP tree and the target value. In particular the "hits" is only used for reporting. The best fitness found in the first run with a population of 48 is given by robust trees which always return a midpoint value (cf. Figure 8) which only passes close to four test points. Trees which closely matched more test points were discovered in the first nineteen generation of this run. However, in terms of fitness, they scored worse than a constant and so went extinct.

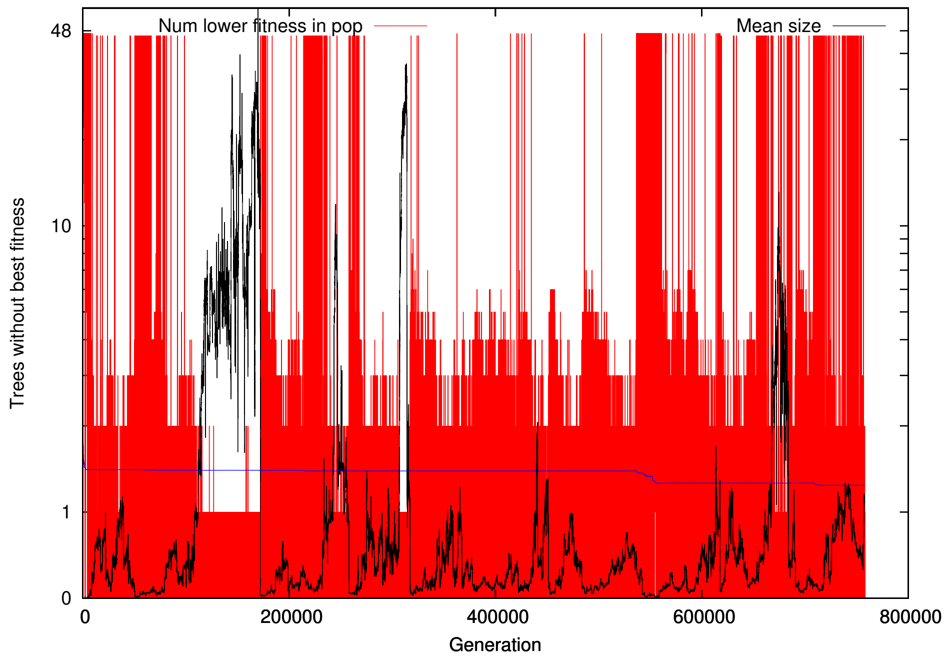

Figure 29: Fitness convergence in last Sextic polynomial run with population of 48 trees. (Run 295 in Figure 2.) In this run 67% of generations all 48 trees have identical fitness. The last improvement is found in generation 752 071 (5759 generations before the end of the run) in a tree of 13 196 331 nodes. It takes over the whole population in 4 generations. However in 21% of the remaining generations it does not totally dominate (mean number of trees with lower fitness 0.394 per generation). Mean prog size (linear scale, black) and best fitness (log, blue).

5 Is there a Limit to Evolution?

In the Sextic Polynomial experiments with larger populations there is no hint of either evolution of fitness or bloat totally stopping. In the smaller populations, it is both possible to run evolution for longer and to allow trees to be even larger. Four of the eleven pop=48 runs reached a million generations but in the remaining seven, bloat ran into memory limits and halted the run. Only in one run did we see anti-bloat, in which the population converged in a few generations on a small high fitness tree which crossover was able to replicate across a million generations. Interestingly two other runs found similar solutions but after thousands of generations crossover found bloated versions of them.

In the binary 6-Mux Boolean problem (Langdon, 2017) there are only 65 different fitness values. Therefore the number of fitness improvements is very limited. An end to bloat was found. By which we mean it was possible for trees to grow so large that crossover was unable to disrupt the important part of their calculation next to the root node and many generations were evolved where everyone had identical fitness. This led to random selection and random fluctuations in tree size, i.e. enormous trees but without a tendency for progressive endless growth.

This did not happen here (except in some of the some of the unsuccessful population 48 runs). Even in some of the smallest Sextic polynomials runs, we are still seeing innovation in the second half of the run, with tiny fitness improvements being created by crossover between enormous parents (Table 4). Also we are still slightly short of total fitness convergence.

However, there is a strong relationship between the size of the population and the success of the runs. All runs of size 4000 were successful, half of the runs of size 500 were successful, but none of the runs of size 48 were successful.

Even with populations containing Sextic polynomial trees of hundreds of millions of nodes, crossover can still be disruptive and frequently even tiny populations can contain a tree of lower fitness. This is sufficient to provide some pressure (over thousands of generations) for tree size to increase on average.

Can bloat continue forever? It is still difficult to be definitive in our answer. We have seen cases where it does not and of course there are plenty of techniques to prevent bloat (Poli and McPhee, 2013). But we see other cases where crossover over thousands of generations can create an innovative child which allows bloat into a converged population of small trees. Perhaps more interestingly, we see crossover finding fitness improvement in bloated trees after many thousand of generations.

It is still an open question in continuous domains if, given sufficient memory and computational resources, bloat will always stifle innovation so completely that crossover will always only reproduce children of exactly the same fitness for long enough that the lack of selection pressure (Langdon and Poli, 1997) will in turn stifle bloat. However the success of incremental evolution implies most of the time even large changes cannot percolate through many functions, even though they are floating point functions which lose little entropy (Langdon et al., 2021a; Petke et al., 2021; Langdon et al., 2021b). We next turn to the implications of the information dissipation caused by heavily nesting function calls.

6 Evolution of Open Complexity

The success of incremental evaluation (Section 2.5) has some profound implications for the evolution of complex programs.

While introduced as a speed-up mechanism which does not alter the course of evolution (Langdon, 2021b), incremental evaluation shows the progressive concealing of, even large, effects by long chains of computation (Langdon et al., 2021a). The inescapable loss of information (Shannon and Weaver, 1964) in nested expressions is the underlying and unifying cause for several widespread features in computing, such as: in testing (Voas, 1992) (Androutsopoulos et al., 2014), failed disruption propagation (Petke et al., 2021), equivalent mutants (Langdon et al., 2010), neutral networks (Harrand et al., 2019), mutational robustness (Schulte et al., 2014), coincidental correctness (Abou Assi et al., 2019), correctness attraction (Danglot et al., 2018), robustness (Langdon and Petke, 2015) and AntiFragile software (Monperrus, 2017), which currently occupy different research silos (Petke et al., 2021).

Incremental evaluation finds the fitness of children starting from the changed code. Because it keeps track of both the evaluation of the changed code and of the original, it is easy to find cases where they are the same. Even for large syntactic or semantic changes, if they are sufficiently deeply buried, incremental evaluation often shows that the run time evaluation of the modified code is identical to that of the original code. That is, the (deep) crossover, mutation, run time perturbation, glitch, error, etc., etc., has no impact.

It appears non-linear, coarse and binary operations lose information faster, i.e. are more dissipative, and so are better at concealing bugs, mutations and training updates than linear or smooth operators.

From the point of view of evolving complex systems, it is not sufficient for an organism to be large. Indeed, instead of deep complexity, we shall need complex systems to be open, allowing changes to percolate out to their environment. Perhaps biologically plausible but more directed crossover or mutation operators, perhaps taking code utility or information throughput into account, could be used. However if mutations are randomly scattered in a large system, there is a risk their influence will have to pass through many layers to reach the environment and thus impact fitness. Instead we might want such a system to be "lung like", with many passageways, allowing the run time impact of mutational changes ready access to the outside. Similarly we might view the evolving organism as a "small world" network which permits the impact of code updates or training events to pass through only a small number of nodes before reaching an external view point. For instance, Hu et al. (2020) consider Boolean circuits with 256 phenotypes showing the statistical properties of their genotype-phenotype maps. Unless there are short paths, the system risks becoming onion-like: where perhaps inner layers once evolved but are now encased in newer outer layers and now adaptation occurs only in the outermost layers. Indeed, the inner layers could congeal into a static lifeless mass, like a planetary core, with learning only occurring on the outermost crust.

7 Conclusions

Evolving Sextic polynomial trees for up to a million generations, during which some programs grow to two billion nodes, suggests even a simple genetic programming (GP) floating point benchmark allows long-term fitness improvement over thousands of generations.

The availability of multi-core SIMD capable hardware has allowed us to push genetic programming performance on single computers with floating point problems to that previously only approached with sub-machine code GP operating in discrete domains (Poli and Langdon, 1999; Poli and Page, 2000). This in turn has allowed GP runs far longer than anything previously attempted whilst evolving far bigger programs.

Without size or depth limits or biases crossover with brutal selection pressure tends to evolve very large non-parsimonious programs, known in the GP community as bloat (Koza, 1992, page 617). (See also footnote 2 on third page.) After a few initial generations, GP tree bloat typically follows a sub-quadratic power law (Langdon, 2000a). But eventually effective selection pressure (Nordin, 1997, sec. 14.2), (Banzhaf et al., 1998, page 187), (Stephens and Waelbroeck, 1999; Langdon and Poli, 2002) within highly evolved populations falls, leading to bloat at a reduced rate. However in this continuous domain we only see the chaotic lack of bloat found in long-running Boolean problems (Langdon, 2017) in a few unsuccessful runs with tiny populations (red plots in Figure 2). Nevertheless in all cases bloated binary trees evolve to be randomly shaped and lie close to Flajolet's square root limit.

Information theory (particularly the inevitable entropy loss) and our experiments with huge nested tree expressions, lead us to conclude that: without short cuts, highly nested routines are robust and resistant to innermost changes. And so we suggest the opposite: large evolvable organisms will have to be open complex systems with many short paths rapidly connecting some of the learning, adaptation or mutation sites to the environment.

Acknowledgements

This work was inspired by conversations at Dagstuhl Seminar 18052 on Genetic Improvement of Software (Petke et al., 2018).

The new parallel GPquick code is available via `http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/gp-code/GPavx.tar.gz`

References

Abou Assi, R., Trad, C., Maalouf, M., and Masri, W. (2019). Coincidental correctness in the Defects4J benchmark. *Software Testing, Verification and Reliability*, 29(3):e1696. `http://dx.doi.org/10.1002/stvr.1696`

Altenberg, L. (1994). The evolution of evolvability in genetic programming. In Kinnear, Jr., K. E., editor, *Advances in Genetic Programming*, chapter 3, pages 47–74. MIT Press. `http://dynamics.org/~altenber/PAPERS/EEGP/`

Androutsopoulos, K., Clark, D., Dan, H., Hierons, R. M., and Harman, M. (2014). An analysis of the relationship between conditional entropy and failed error propagation in software testing.

In Briand, L. and van der Hoek, A., editors, 36^{th} *International Conference on Software Engineering (ICSE 2014)*, pages 573–583, Hyderabad, India. ACM. http://dx.doi.org/10.1145/2568225.2568314

Angeline, P. J. (1994). Genetic programming and emergent intelligence. In Kinnear, Jr., K. E., editor, *Advances in Genetic Programming*, chapter 4, pages 75–98. MIT Press. http://cognet.mit.edu/sites/default/files/books/9780262277181/pdfs/9780262277181_chap4.pdf

Baker, J. E. (1987). Reducing bias and inefficiency in the selection algorithm. In Grefenstette, J. J., editor, *Proceedings of the Second International Conference on Genetic Algorithms and their Application*, pages 14–21, Cambridge, MA, USA. Lawrence Erlbaum Associates.

Banzhaf, W., Nordin, P., Keller, R. E., and Francone, F. D. (1998). *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, San Francisco, CA, USA. https://www.amazon.co.uk/Genetic-Programming-Introduction-Artificial-Intelligence/dp/155860510X

Danglot, B., Preux, P., Baudry, B., and Monperrus, M. (2018). Correctness attraction: a study of stability of software behavior under runtime perturbation. *Empirical Software Engineering*, 23(4):2086–2119. http://dx.doi.org/10.1007/s10664-017-9571-8

Evans, A. R., Jones, D., Boyer, A. G., Brown, J. H., Costa, D. P., Ernest, S. M., Fitzgerald, E. M., Fortelius, M., Gittleman, J. L., Hamilton, M. J., et al. (2012). The maximum rate of mammal evolution. *Proceedings of the National Academy of Sciences*, 109(11):4187–4190. http://dx.doi.org/10.1073/pnas.1120774109

Fernandez de Vega, F., Olague, G., Lanza, D., Chavez de la O, F., Banzhaf, W., Goodman, E., Menendez-Clavijo, J., and Martinez, A. (2020). Time and individual duration in genetic programming. *IEEE Access*, 8:38692–38713. http://dx.doi.org/10.1109/ACCESS.2020.2975753

Harrand, N., Allier, S., Rodriguez-Cancio, M., Monperrus, M., and Baudry, B. (2019). A journey among Java neutral program variants. *Genetic Programming and Evolvable Machines*, 20(4):531–580. http://dx.doi.org/10.1007/s10710-019-09355-3

Hu, T., Tomassini, M., and Banzhaf, W. (2020). A network perspective on genotype-phenotype mapping in genetic programming. *Genetic Programming and Evolvable Machines*, 21(3):375–397. Special Issue: Highlights of Genetic Programming 2019 Events. http://dx.doi.org/10.1007/s10710-020-09379-0

Keith, M. J. and Martin, M. C. (1994). Genetic programming in C++: Implementation issues. In Kinnear, Jr., K. E., editor, *Advances in Genetic Programming*, chapter 13, pages 285–310. MIT Press. http://cognet.mit.edu/sites/default/files/books/9780262277181/pdfs/9780262277181_chap13.pdf

Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA. http://mitpress.mit.edu/books/genetic-programming

Koza, J. R. (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge Massachusetts. http://www.genetic-programming.org/gpbook2toc.html

Koza, J. R., Andre, D., Bennett III, F. H., and Keane, M. (1999). *Genetic Programming III: Darwinian Invention and Problem Solving*. Morgan Kaufmann. http://www.genetic-programming.org/gpbook3toc.html

Langdon, W. B. (1998). *Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming!*, volume 1 of *Genetic Programming*. Kluwer, Boston. http://dx.doi.org/10.1007/978-1-4615-5731-9

Langdon, W. B. (1999a). Linear increase in tree height leads to sub-quadratic bloat. In Haynes, T., Langdon, W. B., O'Reilly, U.-M., Poli, R., and Rosca, J., editors, *Foundations of Genetic Programming*, pages 55–56, Orlando, Florida, USA. `http://www.cs.ucl.ac.uk/staff/W.Langdon/fogp/WBL.fogp.ps.gz`

Langdon, W. B. (1999b). Scaling of program tree fitness spaces. *Evolutionary Computation*, 7(4):399–428. `http://dx.doi.org/10.1162/evco.1999.7.4.399`

Langdon, W. B. (2000a). Quadratic bloat in genetic programming. In Whitley, D., Goldberg, D., Cantu-Paz, E., Spector, L., Parmee, I., and Beyer, H.-G., editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)*, pages 451–458, Las Vegas, Nevada, USA. Morgan Kaufmann. `http://gpbib.cs.ucl.ac.uk/gecco2000/GA069.pdf`

Langdon, W. B. (2000b). Size fair and homologous tree genetic programming crossovers. *Genetic Programming and Evolvable Machines*, 1(1/2):95–119. `http://dx.doi.org/10.1023/A:1010024515191`

Langdon, W. B. (2013). Large scale bioinformatics data mining with parallel genetic programming on graphics processing units. In Tsutsui, S. and Collet, P., editors, *Massively Parallel Evolutionary Computation on GPGPUs*, Natural Computing Series, chapter 15, pages 311–347. Springer. `http://dx.doi.org/10.1007/978-3-642-37959-8_15`

Langdon, W. B. (2017). Long-term evolution of genetic programming populations. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '17, pages 235–236, Berlin. ACM. `http://dx.doi.org/10.1145/3067695.3075965`

Langdon, W. B. (2019). Parallel GPQUICK. In Doerr, C., editor, *GECCO '19: Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 63–64, Prague, Czech Republic. ACM. `http://dx.doi.org/10.1145/3319619.3326770`

Langdon, W. B. (2020a). Genetic improvement of genetic programming. In Brownlee, A. S., Haraldsson, S. O., Petke, J., and Woodward, J. R., editors, *GI @ CEC 2020 Special Session*, page paper id24061, internet. IEEE Computational Intelligence Society, IEEE Press. `http://dx.doi.org/10.1109/CEC48606.2020.9185771`

Langdon, W. B. (2020b). Multi-threaded memory efficient crossover in C++ for generational genetic programming. *SIGEVOLution newsletter of the ACM Special Interest Group on Genetic and Evolutionary Computation*, 13(3):2–4. `http://dx.doi.org/10.1145/3430913.3430914`

Langdon, W. B. (2020c). Multi-threaded memory efficient crossover in C++ for generational genetic programming. ArXiv. `http://arxiv.org/abs/2009.10460`

Langdon, W. B. (2021a). Fitness first. In Banzhaf, W., Trujillo, L., Winkler, S., and Worzel, B., editors, *Genetic Programming Theory and Practice XVIII*, Genetic and Evolutionary Computation, pages 143–164, East Lansing, MI, USA. Springer. `http://dx.doi.org/10.1007/978-981-16-8113-4_8`

Langdon, W. B. (2021b). Incremental evaluation in genetic programming. In Hu, T., Lourenco, N., and Medvet, E., editors, *EuroGP 2021: Proceedings of the 24th European Conference on Genetic Programming*, volume 12691 of *LNCS*, pages 229–246, Virtual Event. Springer Verlag. `http://dx.doi.org/10.1007/978-3-030-72812-0_15`

Langdon, W. B. (2022). Genetic programming convergence. *Genetic Programming and Evolvable Machines*. `http://dx.doi.org/10.1007/s10710-021-09405-9`

Langdon, W. B. and Banzhaf, W. (2008). Repeated patterns in genetic programming. *Natural Computing*, 7(4):589–613. `http://dx.doi.org/10.1007/s11047-007-9038-8`

Langdon, W. B. and Banzhaf, W. (2019). Continuous long-term evolution of genetic programming. In Fuechslin, R., editor, *Conference on Artificial Life (ALIFE 2019)*, pages 388–395, Newcastle. MIT Press. `http://dx.doi.org/10.1162/isal_a_00191`

Langdon, W. B., Harman, M., and Jia, Y. (2010). Efficient multi-objective higher order mutation testing with genetic programming. *Journal of Systems and Software*, 83(12):2416–2430. http://dx.doi.org/10.1016/j.jss.2010.07.027

Langdon, W. B. and Petke, J. (2015). Software is not fragile. In Parrend, P., Bourgine, P., and Collet, P., editors, *Complex Systems Digital Campus E-conference, CS-DC'15*, Proceedings in Complexity, pages 203–211. Springer. Invited talk. http://dx.doi.org/10.1007/978-3-319-45901-1_24

Langdon, W. B. and Petke, J. (2019). Genetic improvement of data gives binary logarithm from sqrt. In Allmendinger, R. et al., editors, *GECCO '19: Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 413–414, Prague, Czech Republic. ACM. http://dx.doi.org/10.1145/3319619.3321954

Langdon, W. B., Petke, J., and Clark, D. (2021a). Dissipative polynomials. In Veerapen, N., Malan, K., Liefooghe, A., Verel, S., and Ochoa, G., editors, *5th Workshop on Landscape-Aware Heuristic Search*, GECCO 2021 Companion, pages 1683–1691, Internet. ACM. http://dx.doi.org/10.1145/3449726.3463147

Langdon, W. B., Petke, J., and Clark, D. (2021b). Information loss leads to robustness. IEEE Software Blog. http://blog.ieeesoftware.org/2021/09/information-loss-leads-to-robustness-w.html

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

Langdon, W. B. and Poli, R. (2002). *Foundations of Genetic Programming*. Springer-Verlag. http://dx.doi.org/10.1007/978-3-662-04726-2

Langdon, W. B., Soule, T., Poli, R., and Foster, J. A. (1999). The evolution of size and shape. In Spector, L., Langdon, W. B., O'Reilly, U.-M., and Angeline, P. J., editors, *Advances in Genetic Programming 3*, chapter 8, pages 163–190. MIT Press, Cambridge, MA, USA. http://www.cs.ucl.ac.uk/staff/W.Langdon/aigp3/ch08.pdf

Lenski, R. E. (1988). Experimental studies of pleiotropy and epistasis in Escherichia coli. I. Variation in competitive fitness among mutants resistant to virus T4. *Evolution*, 42(3):425–432. http://dx.doi.org/10.1111/j.1558-5646.1988.tb04149.x

Lenski, R. E. et al. (2015). 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). http://dx.doi.org/10.1098/rspb.2015.2292

McPhee, N. F. and Poli, R. (2001). 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., and Langdon, W. B., editors, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 108–125, Lake Como, Italy. Springer-Verlag. http://dx.doi.org/10.1007/3-540-45355-5_10

Monperrus, M. (2017). Principles of antifragile software. In *Companion to the First International Conference on the Art, Science and Engineering of Programming*, Programming '17, pages 32:1–32:4, New York, NY, USA. ACM. http://dx.doi.org/10.1145/3079368.3079412

Nordin, P. (1997). *Evolutionary Program Induction of Binary Machine Code and its Applications*. PhD thesis, der Universitat Dortmund am Fachereich Informatik, Germany. http://www.amazon.co.uk/Evolutionary-Program-Induction-Machine-Applications/dp/3931546071

Owen, R. B., Crossley, R., Johnson, T. C., Tweddle, D., Kornfield, I., Davison, S., Eccles, D. H., and Engstrom, D. E. (1990). 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. http://www.jstor.org/stable/49477

Palumbo, S. (2001). *The Evolution Explosion*. Norton.

Petke, J., Clark, D., and Langdon, W. B. (2021). Software robustness: A survey, a theory, and some prospects. In Avgeriou, P. and Zhang, D., editors, *ESEC/FSE 2021, Ideas, Visions and Reflections*, pages 1475–1478, Athens, Greece. ACM. http://dx.doi.org/10.1145/3468264.3473133

Petke, J., Le Goues, C., Forrest, S., and Langdon, W. B. (2018). Genetic improvement of software: Report from Dagstuhl Seminar 18052. *Dagstuhl Reports*, 8(1):158–182. http://dx.doi.org/10.4230/DagRep.8.1.158

Poli, R. and Langdon, W. B. (1999). Sub-machine-code genetic programming. In Spector, L., Langdon, W. B., O'Reilly, U.-M., and Angeline, P. J., editors, *Advances in Genetic Programming 3*, chapter 13, pages 301–323. MIT Press, Cambridge, MA, USA. http://www.cs.ucl.ac.uk/staff/W.Langdon/aigp3/ch13.pdf

Poli, R., Langdon, W. B., and McPhee, N. F. (2008). *A field guide to genetic programming*. Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk. (With contributions by J. R. Koza). http://www.gp-field-guide.org.uk

Poli, R. and McPhee, N. F. (2013). Parsimony pressure made easy: Solving the problem of bloat in GP. In Borenstein, Y. and Moraglio, A., editors, *Theory and Principled Methods for the Design of Metaheuristics*, Natural Computing Series, pages 181–204. Springer. http://dx.doi.org/10.1007/978-3-642-33206-7_9

Poli, R. and Page, J. (2000). 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. http://dx.doi.org/10.1023/A:1010068314282

Schulte, E., Fry, Z. P., Fast, E., Weimer, W., and Forrest, S. (2014). Software mutational robustness. *Genetic Programming and Evolvable Machines*, 15(3):281–312. http://dx.doi.org/10.1007/s10710-013-9195-8

Sedgewick, R. and Flajolet, P. (1996). *An Introduction to the Analysis of Algorithms*. Addison-Wesley.

Shannon, C. E. and Weaver, W. (1964). *The Mathematical Theory of Communication*. The University of Illinois Press, Urbana, Il, USA. http://www.press.uillinois.edu/books/catalog/67qhn3ym9780252725463.html

Singleton, A. (1994). Genetic programming with C++. *BYTE*, pages 171–176. http://www.assembla.com/wiki/show/andysgp/GPQuick_Article

Stephens, C. and Waelbroeck, H. (1999). Schemata evolution and building blocks. *Evolutionary Computation*, 7(2):109–124. http://dx.doi.org/10.1162/evco.1999.7.2.109

Syswerda, G. (1990). A study of reproduction in generational and steady state genetic algorithms. In Rawlings, G. J. E., editor, *Foundations of genetic algorithms*, pages 94–101. Morgan Kaufmann, Indiana University. Published 1991. http://dx.doi.org/10.1016/B978-0-08-050684-5.50009-4

Tackett, W. A. (1994). *Recombination, Selection, and the Genetic Construction of Computer Programs*. PhD thesis, University of Southern California, Department of Electrical Engineering Systems, USA. http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/ftp.io.com/papers/WAT_PHD_DissFull_USC94_Recombination_etc_Genetic_Construction_of_Computer_Programs.pdf

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