Continuous Long-Term Evolution of 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. Programs with almost 400 000 000 instructions are created by crossover. To support unbounded Long-Term Evolution Experiment LTEE GP we use both SIMD parallel AVX 512 bit instructions and 48 threads to yield performance of up to 149 billion GP operations per second, 149 giga GPops, on a single Intel Xeon Gold 6126 2.60 GHz server.

Introduction

Nature has had billions of years for evolution to work its way to the organisms we see today. Not only was a long time available to achieve results, but many generations passed before the present. In evolutionary biology, there is discussion about the long-term innovative capabilities of evolution. Some say, 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 maintain 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 for a long time, and are not disrupted. 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, 1,000, 10,000 etc. generations of continued evolution? Does it stagnate? Does it continue to produce surprises?

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 generations being frozen every so often 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). In 2019, 70,000 generations have been reached, with no end to evolution in sight.

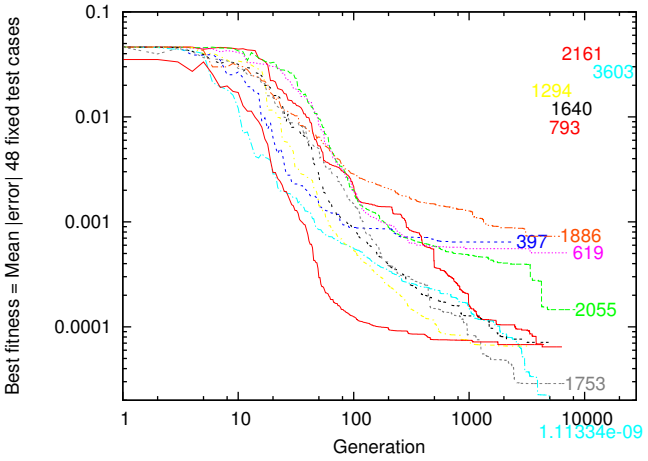

Figure 1: Evolution of mean absolute error in ten runs of Sextic polynomial (Koza, 1992) with population of 4000. (Runs aborted after first crossover to hit 15 million node limit.) End of run label gives number of generations when fitness got better (five shown at top right to avoid crowding).

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. Excluding some special applications or Boolean benchmarks based on graphics hardware (GPUs), at up to

149 billion GP operations per second (149 giga-GPops, see Table 3), this appears to be the fastest single-computer GP system (Langdon, 2013, Tab. 3).

In the Boolean domain we found usually 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. Figure 1 shows that although the rate of innovation falls (as in Lenski's *E. Coli*[1] populations), 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 sub-tree 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[2], and so we need a GP system not only able to run for a million generations[3] but also able to process trees with well in excess of a 100 million nodes[4]. 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.

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, 1999) and Flajolet limit (depth $\approx \sqrt{2\pi|\text{size}|}$ (Langdon, 2000b)) to essentially hold. More analysis can be found in the technical report (Langdon and Banzhaf, 2019). 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.

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) and then how GPQUICK has been modified to run in parallel.

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 (1992) shows doubling is not necessary). 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

[1] The *E. Coli* genome contains 4.6 million DNA base pairs.

[2] 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). 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.

[3] 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.

[4] Again referring to the extended runs in Figure 2, crossover creates highly evolved trees containing almost four hundred million nodes. These are by far the largest programs yet evolved.

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

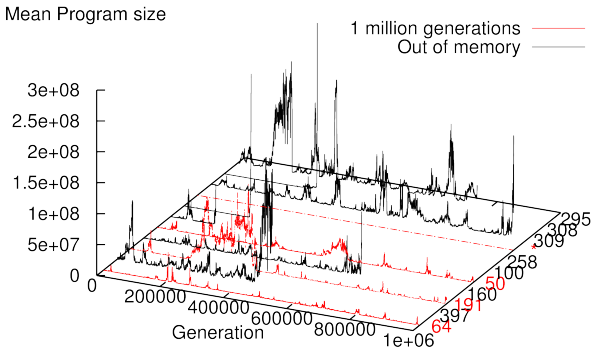

Figure 2: 11 extended runs pop=48. Numbers on right indicate size of largest tree before the run stopped in millions of nodes. One run (*) converged so that more than 90% of the trees contain just five nodes. Three of the other four runs that reached 1 million generations (red) took between half a day 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).

($+$, $-$, $\times$ and protected division), an input (x) and 250 constants. The constants are chosen uniformly at random from the 2001 floating point numbers from -1.000 to +1.000. 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.

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).

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. 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 Figure 4.) Thus the user-specified tree size limit can 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. Additional details can be found in Langdon and Banzhaf (2019).

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. 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 so that this cannot affect the course of evolution. E.g., pseudo random numbers are only generated in sequential code.

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.

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 48 C++ pthreads. The results are identical but give an additional ≈two-fold speed up.

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 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 systems 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.

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`

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 long 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.

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 (previous page), 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

Table 2: 10 Sextic polynomial runs with population 4000

Gens	err10^{-9}	impr[6]	hits	size10^6	x^i	conv	ops10^9
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

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

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.

Results

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 but one 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 3 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 highly evolved trees tend to be randomly shaped and so as expected

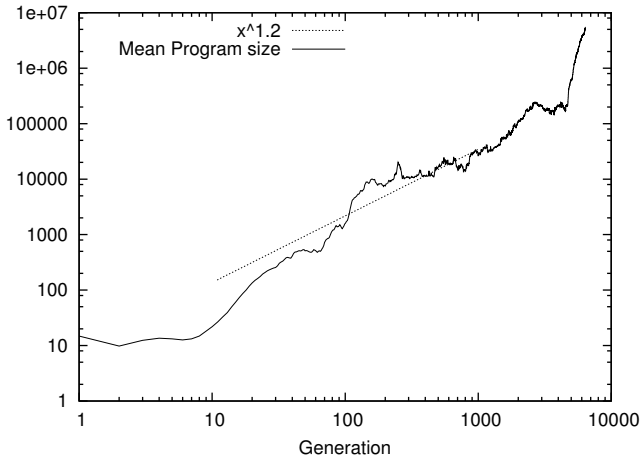

Figure 3: Evolution of tree size in first Sextic run (population 4000). (This run aborted after 6370 generations by first crossover to hit 15 million node limit.) Straight line shows best RMS error power law fit between generation 10 and 1000, $y = 8.65x^{1.2001}$

tend to lie near the Flajolet limit, depth $\approx \sqrt{2\pi|\text{size}|}$ (see Figure 4). (This is also true in the pop=500 and pop=48 runs, see following sections.)

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 5), 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. 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 5 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).

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 6 of these runs. Notice two runs do not really solve the problem and only hit less than half the test cases (see "hits" column in Table 3). 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).

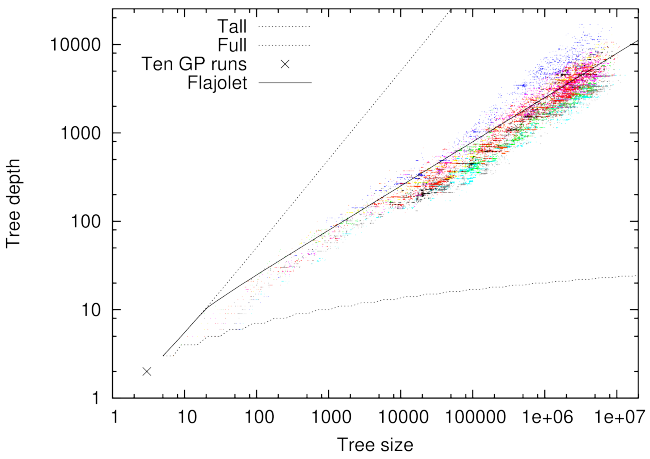

Figure 4: Plot of size and depth of the best individual in each generation for 10 Sextic polynomial runs with population of 4000. Binary trees must lie between short fat trees (lower curve "Full") and "Tall" stringy trees. Most trees are randomly shaped and lie near the Flajolet limit (depth $\approx\sqrt{2\pi|\text{size}|}$, solid line, note log-log scales).

Table 3: 6 Sextic polynomial runs with population 500

Gens	err10^{-6}	impr	hits	size10^6	x^i	conv	ops10^9
111582	538	3545	47	399.594	1.558	500	93.8
23937	34313	757	18	202.439	1.736	500	117.3
35783	307	3484	48	227.488	1.436	500	95.8
43356	18373	929	22	267.416	2.181	500	149.2
27713	137	5852	48	327.253	1.928	500	138.9
103953	1765	664	48	230.106	1.408	500	69.6

Since we have deliberately extended the space available to GP trees, it is no surprise that the trees grow even bigger than before (column 5 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)).

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). For example, in the first Sextic polynomial pop=500 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. 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.

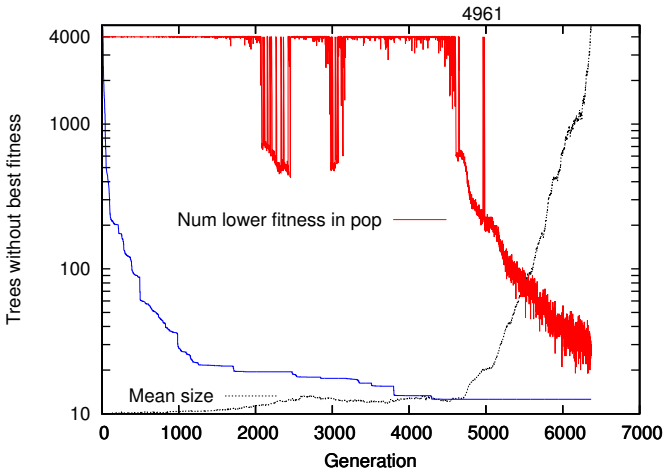

Figure 5: Fitness convergence in first Sextic polynomial pop=4000 run. Perhaps because of the continual discovery of better trees before generation 4975 and the larger population size, although the number of tree without the best fitness falls, unlike in the earlier Boolean problem (Langdon, 2017) it 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.)

Results Population 48 trees

In the final experiments the population was reduced still further to allow even larger trees to be evolved (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 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. 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.

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 (Figure 2). The best fitness found in this run is given by robust trees which always return a midpoint value 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.

Table 4: 11 Sextic polynomial runs with population of 48

Gens	err10^{-6}	impr	hits	size10^6	x^i	conv	ops10^9
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

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 version 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. 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. 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.

Conclusions

Evolving binary Sextic polynomial trees for up to a million generations, during which some programs grow to four hundred million nodes, suggests even a simple 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 GP 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 second 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.

Acknowledgements

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

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

References

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.

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.

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.

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.

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.

Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press.

Koza, J. R. (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press.

Langdon, W. B. (1998). *Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming!*, volume 1 of *Genetic Programming*. Kluwer.

Langdon, W. B. (1999). Linear increase in tree height leads to subquadratic bloat. In Haynes, T. et al., editors, *Foundations of Genetic Programming*, pages 55–56, Orlando, Florida, USA.

Langdon, W. B. (2000a). Quadratic bloat in genetic programming. In Whitley, D. et al., editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)*, pages 451–458, Las Vegas, Nevada, USA. Morgan Kaufmann.

Langdon, W. B. (2000b). Size fair and homologous tree genetic programming crossovers. *Genetic Programming and Evolvable Machines*, 1(1/2):95–119.

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.

Langdon, W. B. (2017). Long-term evolution of genetic programming populations. In *GECCO 2017: The Genetic and Evolutionary Computation Conference*, pages 235–236, Berlin.

Langdon, W. B. and Banzhaf, W. (2019). Faster genetic programming GPquick via multicore and advanced vector extensions. Technical Report RN/19/01, University College, London, London, UK.

Langdon, W. B. and Poli, R. (1997). Fitness causes bloat. In Chawdhry, P. K. et al., editors, *Soft Computing in Engineering Design and Manufacturing*, pages 13–22. Springer-Verlag London.

Langdon, W. B. and Poli, R. (2002). *Foundations of Genetic Programming*. Springer-Verlag.

Langdon, W. B., Soule, T., Poli, R., and Foster, J. A. (1999). The evolution of size and shape. In Spector, L. et al., editors, *Advances in Genetic Programming 3*, chapter 8, pages 163–190. MIT Press.

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*, 42:425–432.

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).

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. et al., editors, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 108–125, Lake Como, Italy. Springer-Verlag.

Nordin, P. (1997). *Evolutionary Program Induction of Binary Machine Code and its Applications*. PhD thesis, der Universitat Dortmund am Fachereich Informatik.

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.

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

Poli, R. and Langdon, W. B. (1999). Sub-machine-code genetic programming. In Spector, L. et al., editors, *Advances in Genetic Programming 3*, chapter 13, pages 301–323. MIT Press.

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).

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.

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.

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

Singleton, A. (1994). Genetic programming with C++. *BYTE*, pages 171–176.

Stephens, C. and Waelbroeck, H. (1999). Schemata evolution and building blocks. *Evolutionary Computation*, 7(2):109–124.

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.

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.