Chapter 1
Long Term Evolution Experiments with Linear Genetic Programming

William B. Langdon, Computer Science, University College, London, UK

Abstract Inspired by Richard Lenski's Long-Term Evolution Experiment, we use the quantised chaotic Mackey-Glass time series as a prolonged learning task for artificial evolution in the form of steady state linear genetic programming using multi-threaded AVX512 GPengine to reach 100 000 generations, 4 million arithmetic instructions and speeds of up to the equivalent of 361 billion GP operations per second (361 10^9 GPops^{-1}) on a 3.1 GHz multi core computer. Typically finding hundreds of fitness improvements in the later stages of the runs. Long fit programs are typically robust to two point crossover and random point mutation. They loose entropy monotonically towards the entropy of the fitness target. However almost all their instructions, despite not being reversible, are isentropic, i.e. do not loose entropy, and instead shuffle information between registers.

Keywords: Autonomous open-ended learning in machines, LTEE, time series prediction, Voas PIE, information theory, failed disruption propagation, FDP, adiabatic irreversible arithmetic, population convergence.

1.1 Introduction

Rich Lenski's 38 year Long-Term Evolution Experiment [Lenski and others, 2015] has shown, even in stable environments, bacteria can continue to evolve, even after 82 500 generations. (In contrast Homo Sapiens is some 9300 generations old[1].) Previously we have asked the question what happens if we allow artificial evolution, specifically tree genetic programming (GP) [Koza, 1992; Poli *et al.*, 2008], to

Revision : 1.75 *Forthcoming* Linear Genetic Programming, Wolfgang Banzhaf and Ting Hu *Eds.*

[1] [Wang *et al.*, 2023] show that the length of human generations has changed over time but estimate on average it has been 26.9 years. Estimates of when the Homo Sapiens species started vary but 250 000 years ago seems reasonable. Giving about 9 300 generations in total.

evolve for tens of thousands, even hundreds of thousands of generations [Langdon and Banzhaf, 2022]. Whilst we found evolution continued, in purely hierarchical tree GP using only crossover, we found the rate of innovation fell inversely in proportion to program size due to failed disruption propagation [Petke *et al.*, 2021; Langdon and Clark, 2024; Langdon and Clark, 2025] giving robustness [Langdon and Petke, 2015] and promoting convergence of populations of GP trees [Langdon, 2022a]. Information theory shows failed disruption propagation is inherent in digital computing and in deep programs can quickly lead to almost all changes (good or bad) being invisible, and so evolution simply drifting and innovation stalling.

Instead we have tried to promote the idea that, to avoid failed disruption propagation stifling evolution, for long term innovation we need to evolve thin walled software with a high surface area (such as inspired by human lungs [Langdon, 2022b]). We wish to ensure that fitness disruption in the bulk of the code (where most genetic operations will act) has only a short distance to travel to the surrounding environment and so is likely to be visible and so beneficial changes can be seen and rewarded (and negative changes be seen and punished) [Langdon and Hulme, 2024]. Like chemical reactions occurring on a catalyst's surface, rather than membranes or skins separating computing regions, we want computing to occur on such surfaces.

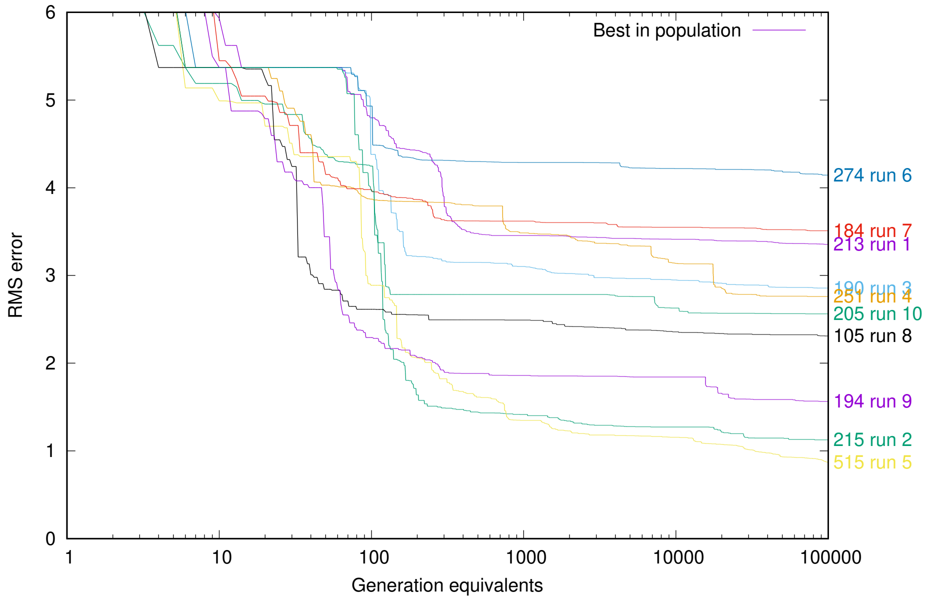

Fig. 1.1 Evolution of best fitness in ten runs of discrete Mackey-Glass chaotic sequence prediction with population of 500. Number of fitness improvements given at the end of each run's trace. (Note the similarity with some biological fitness measures in Lenski's LTEE, e.g. [Chihoub *et al.*, 2025, Fig 1A].) The same colours are used for the ten runs in Figures 1.7, 1.8, 1.10, 1.9, 1.11, 1.12, 1.13 and 1.22.

Our intention is to investigate other evolving architectures. We start with linear genetic programming [Nordin, 1994; Banzhaf *et al.*, 1998; Brameier and Banzhaf, 2007] (Figure 1.1) but will in future investigate evolution of arrays or networks of such programs. We also swap from continuous (float) symbolic regression to predicting discrete (integer) time series, deliberately choosing a chaotic series, as it should prove hard enough to continually challenge evolution. Indeed the Mackey-Glass series (Figures 1.2 and 1.3) can be extended should the predictor approach solving any finite part of it.

We do not want to impose arbitrary limits but it must be admitted that without size control we expect bloat [Koza, 1992; Tackett, 1994; Langdon and Poli, 1997; Altenberg, 1994; Angeline, 1994; Banzhaf and Langdon, 2002; Poli and McPhee, 2013]. Therefore we need a GP system not only able to run for perhaps a million generations but also able to cope with programs of well in excess of a million nodes.

The following sections (1.2–1.5.4) describe our linear genetic programming system (GPengine), whilst Section 1.6 describes the evolution up to 100 000 generations. Sections 1.6.2–1.6.5.6 provide detailed analysis of programs at generation 500, particularly in terms of information theory, 1.6.3–1.6.4, and genetic robustness/fragility (Sections 1.6.5/1.6.6). We conclude in Section 1.7 that evolution with a stable fitness function can create closed linear programs of huge resilience which preserve internal variable data distributions for thousands of instructions. Therefore whilst long term evolution is possible with linear GP, large linear programs suffer from the same over stability problem as large evolved trees (although the mechanisms are different); both types of populations converge to become excessively robust and the rate of innovation falls as the programs get bigger.

1.2 Linear Genetic Programming GPengine

GPengine is based on code provided by Peter Nordin (the author of the famous commercial linear genetic programming system Discipulus [Nordin, 1994; Francone, 2001]). In [Langdon and Nordin, 2001] we evolved functions with four outputs while in [Langdon and Banzhaf, 2005] this was reduced to one for the Mackey-Glass prediction problem. GPengine is available via `https://github.com/wblangdon/GPengine`.

1.3 A Hard Benchmark: Mackey-Glass

Whereas previously for our long term tree GP experiments, we had used a well known symbolic regression benchmark, which allows contiguous incremental improvement, here we chose a deliberately difficult discrete (short) integer prediction benchmark.

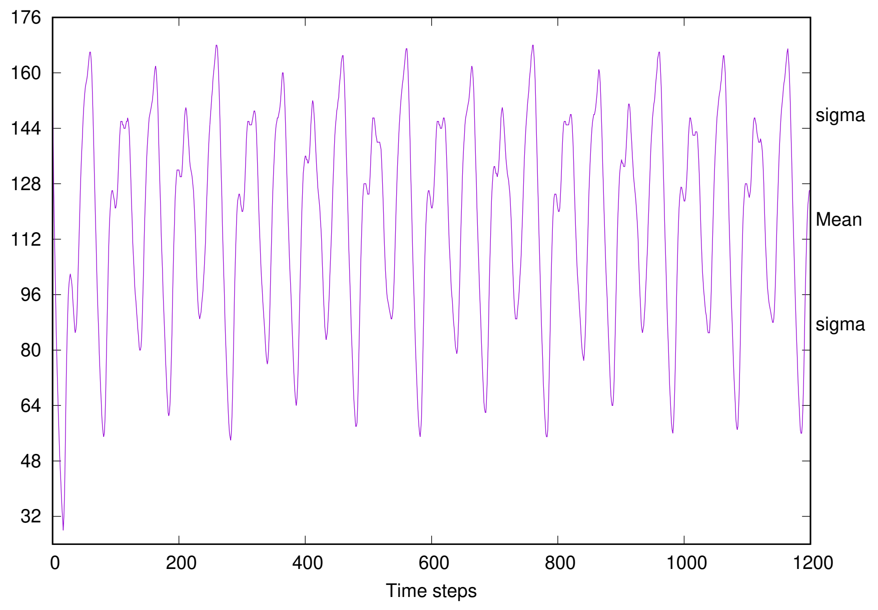

Fig. 1.2 The start of the discrete Mackey-Glass chaotic time series

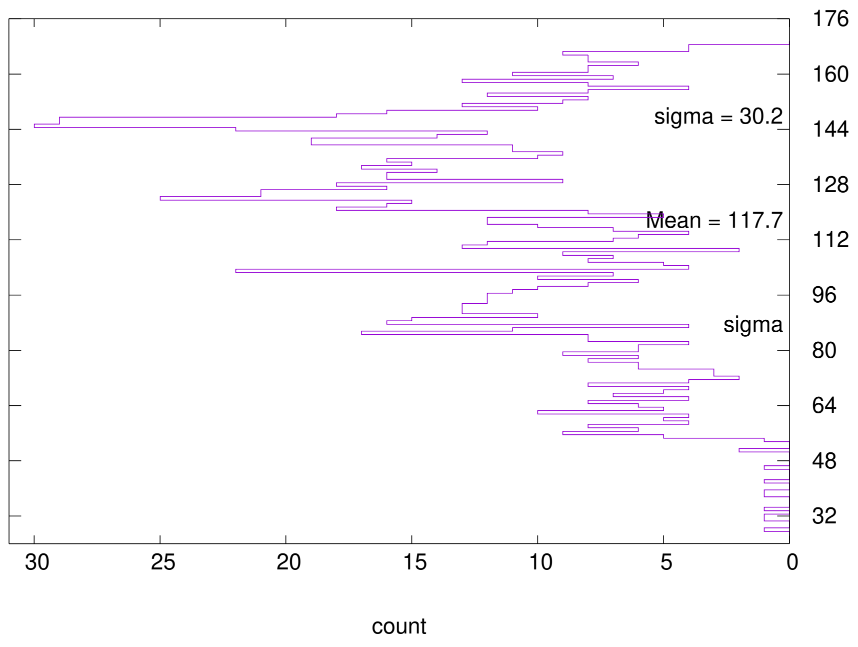

Fig. 1.3 Mackey-Glass benchmark, Figure 1.2, distribution of values. Entropy 6.67 bits.

We used the first 1201 data points of the IEEE benchmark Mackey-Glass chaotic time series, with $\tau = 17$ (Figures 1.2 and 1.3). Being chaotic, the prediction problem should be impossible, nonetheless genetic programming can evolve approximate solutions [Oakley, 1994], and, as we shall see, despite the deliberate granularity of using byte sized integers, linear GP can improve these over time.

Mackey-Glass is a continuous problem. The benchmark converts it to discrete time and we digitised the continuous data to give byte sized integers (by multiplying by 128 and rounding to the nearest integer) [Langdon and Banzhaf, 2005].

The dataset is available via the GPengine GitHub pages and via http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/gp-code/mackey_glass.tar.gz

1.4 Steady State GP, GPengine, Parallel Fitness Evaluation

To contrast with our previous generational tree GP runs (Section 1.1), we use GPengine which is a steady state evolutionary system. In steady state genetic algorithms new children are immediately added to the evolving population and the same number removed, thus keeping the population size constant [Syswerda, 1989]. In nature this can be seen as similar to trees in a temperate forest, where even though the trees may only fruit once a year, the forest contains both seedling and mature trees. Whereas in generational Genetic Algorithms (like our earlier tree GP runs) each population is distinct and when the new generation is formed everyone in the previous generation is killed. In nature, this is like annual plants, which die in the autumn and only their seeds survive the fall and the winter to sprout and new plants grow in the spring.

GPengine uses two point crossover (Figure 1.5). Two fit parents are chosen by binary tournament selection (Table 1.1) and two others are similarly chosen to be removed from the population. In fact they are overwritten by the two newly created children.

The two children are produced one after the other and similarly their fitness' are evaluated one after the other. Notice, here in steady state, fitness is calculated immediately, whereas in a generational GA, the whole population can potentially be processed in parallel together. Therefore GPengine interprets their 1201 fitness cases (see Section 1.3) in parallel using vector instructions [Langdon and Hanna, 2026; Langdon, 2025] and loosely linked Linux ptheads. Up to nineteen 3.1 GHz AMD EPYC 9554 cores and 30 GB memory were used in parallel. (Notice although generational GAs have both a parent and a child population, for the same size of population, they can be implemented to use the same memory size as a steady state GA [Langdon, 2020b; Langdon, 2020c].)

1.5 Experiments

Table 1.1 contains the details of our ten extended linear genetic programming runs with the Mackey-Glass benchmark. As gathering data needed for entropy analysis etc. imposes further runtime overheads, for Section 1.6.2 onwards, where we studied the evolved programs in detail, we re-ran the same runs but only for the first 500 generation equivalents.

Table 1.1 Mackey-Glass prediction with Linear GP

Terminal set:	Unsigned 8 bit integers. Variables `a, b, c, d, e, f, g, h.` Integer constants 0 to 127.
Function set:	$+ - \times$ DIV
Fitness cases:	1201 Mackey-Glass examples. Given 8 prior values (-1, -2, -4, ... -128 steps before) predict next y. See Figure 1.4.
Selection:	Tournament size=2, minimise fitness = $\sum_{i=0}^{1201} \lvert \mathrm{GP}(\mathbf{x_i}) - y_i \rvert^2$
Population:	500, panmictic, steady state.
Parameters:	500 or 100 000 generations. Random initial population (500) size between 1 and 14 instructions. Max 4 000 000 instructions. 90% two point 2 child crossover, 40% chance both XO children subjected to random point mutation 4 times. 10% reproduction.

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

1.5.1 Evolving new programs: Crossover and Elitism

After the initial population has been created, `Evolve` continuously creates pairs of new children. To keep the population constant (`DefaultPopSize = 500`), it also removes two members of the population. `Evolve` calls `Tournament` which starts by selecting uniformly at random four different members of the current population. (Note this means `DefaultPopSize` must be at least 4.) Two binary (two member) tournaments are held to choose two winners and two losers. The winners will be parents and used to create two children, who will replace the two losers.

At random (fraction `PCrossover` = 90%) pairs of children are created by `Crossover` (otherwise `Reproduction` is used, see below). Uniformly at random `Crossover` chooses two crossover cut points in each parent (see top pair of programs in Figure 1.5) and the middle code segments (diagonal hatching) are exchanged to give the two children (lower pair in Figure 1.5).

For efficiency and practicality GPengine imposes a limit (`MaxInstr`) on the length of the evolved programs. If either child would exceed it (4 million instructions), one right hand cut point is deterministically adjusted, so that the new length of the inserted code ensures this child is of maximum length (rather than bigger). That is, `ChooseXO` makes the smallest adjustment to the randomly chosen cut points which is compatible with `MaxInstr`.

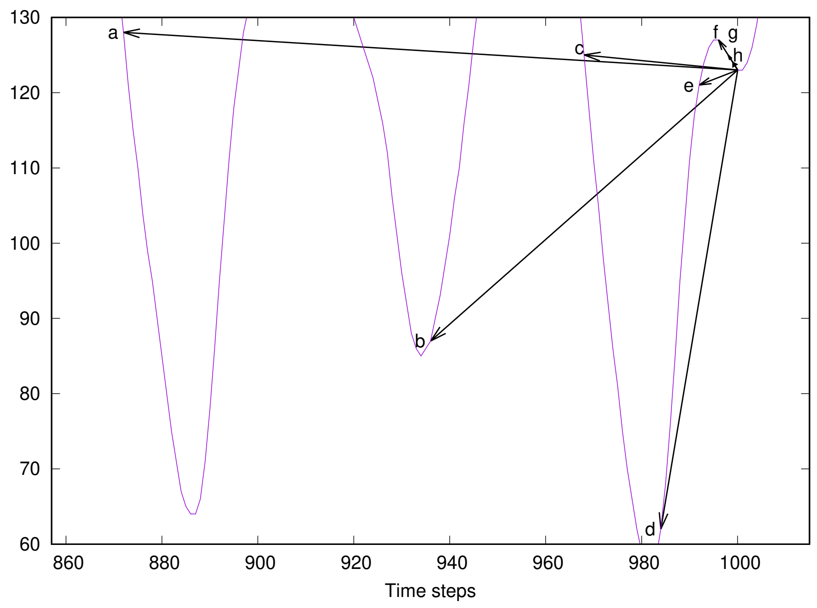

Fig. 1.4 Fragment of Mackey-Glass time series (Figure 1.2). Eight arrow heads (a–h) show source of data for registers a–h used to predict the value at the next timestep (here t=1000). a 128 time steps before (i.e. t=872), b 64, c 32, d 16, e 8, f 4, g 2 and h 1 step before 1000. E.g. in this t=1000 fitness case, register h holds the value (124) at t=999.

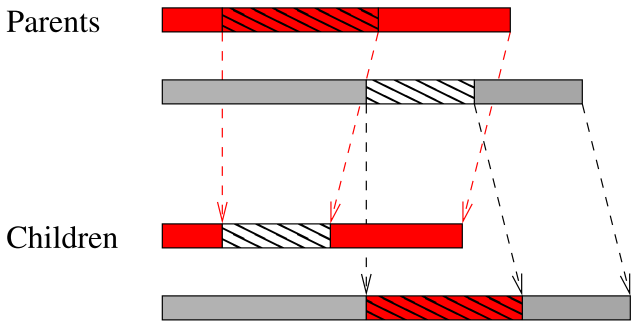

Fig. 1.5 Two crossover points are randomly chosen in each parent (top two genomes) as cut points, to give two children (lower pair). Notice the middle exchanged code (red-pattern and white-pattern) are typically of different sizes, so the two child programs are typically not the same lengths as their parents.

This means if the whole population of programs has bloated to be `MaxInstr` instructions long, the two code fragments exchanged by the parents to create the offspring will be of the same length. However even then, crossover remains true to its original nature and does not become homologous [Nordin *et al.*, 1999; Francone *et al.*, 1999; Langdon, 1999; Hansen, 2003]. Instead, although the crossover fragments are the same length, they are not drawn from the same locations and so we would expect the two children not to be identical to their parents.

Typically with tournament selection in a steady state genetic algorithm fitness will be monotonically improving. That is, the best in population should never get worse. This is because the current best individual will keep winning tournaments and never be selected for replacement. In principle it could be immortal. However this depends on how ties are handled. Usually tournaments where everyone has the same fitness are decided randomly, so even the best individual in the population can die if multiple programs in the population have the best fitness. Therefore steady state GAs can be considered naturally elitist, in the sense that the elite best will always remain in the population until something better evolves.

Nonetheless, in GPengine not all new children are created by crossover and mutation. Instead 10% (i.e. 100 - `PCrossover`) are copies of the winners (elite) of the binary tournaments. `Reproduction` is used to create the remaining ten percent of offspring by copying the two fit winners over writing the two losers. As the children are identical to their parents, there is no need to evaluated their fitness, and instead their fitness values are simply copied from their parents.

1.5.2 Mutation

40 percent of the time both children produced by crossing over two parents are mutated. GPengine uses point mutation, so that four times for each child `Mutation` chooses uniformly at random an instruction (Figure 1.6) to be mutated. As reselection is allowed, potentially an instruction could be chosen more than once, but with long programs this is unlikely. As the programs get longer, there is proportionately less chance for an individual instruction being chosen for mutation. Nonetheless as all instructions are executed, even in long programs, every mutation can potentially impact the child's fitness.

Output a .. h	Arg 1 a .. h	Opcode + − * /	Arg 2 0...127 or a .. h

Fig. 1.6 Format of a GPengine instruction. Notice there are no branches or loops, so effectively all instructions are executed exactly once for each of the 1201 test cases.

To inflict a point mutation, one of the four components of the chosen instruction (Figure 1.6) is chosen uniformly at random. In each case, that part of the instruction is replaced by a new randomly generated component. To ensure the new instruction is syntactically correct, the code that was used to create the initial random population is reused. Only in the cases of the output register (`a, b, c, d, e, f, g` or `h`) and the Opcode ($+, -, \times$, or protected division) does `MutateInstr` ensure that the new field is different from the existing value.

1.5.3 Optimisation, Ineffective Code Removal

As well as executing fitness cases in parallel (Section 1.4), to speed up fitness evaluation we ignore instructions that do not impact the output (given by the final value in register `a`). We follow Peter Nordin's intron removal algorithm. Essentially `Simplify` does two passes. The first scans backwards from the program's end finding the last instruction which overwrote register `a`, noting where it is and extracting the registers (`addactive`) it depends on. Then it continues to scan backwards using `testactive` to find (and record by setting `needed[i]`) the locations of the previous instructions which wrote to them, and so on to the beginning of the program. Notice the set of `active` registers need not expand to include them all because once a register has been over written its previous value does not matter and so `clear` is used to remove it from the `active` set.

For added efficiency, `addactive` checks for some special cases which mean the output of an opcode does not depend upon the values held by the input registers and so does not add them to the `active` set. With our simple function set, the only case where this is true is where a register is subtracted from itself (which always gives zero).

Although it is possible to create and interpret the set of active instructions in reverse order, it is simpler to use the intermediate array (`needed`) to hold the ordered list of active instructions and use a second pass to create, the now shorter, list of active instructions (`Instr2`). Then pass them to the interpreter to be executed in the usual forward direction. Since the cache hardware may assume data is used from start to end, interpreting in the usual direction may be faster (as well as simpler). Remember `Simplify` is only needed once, whereas the program will be interpreted 1201 times (once per fitness test case).

In our runs the fraction of remaining code which is executed is highly variable (see Figure 1.9). With `Simplify`'s intron removal giving between 3.3$\times$ and 700 fold reduction in number of instructions (median 14). This is one factor in the wide range of speeds obtained by GPengine, see Figure 1.7.

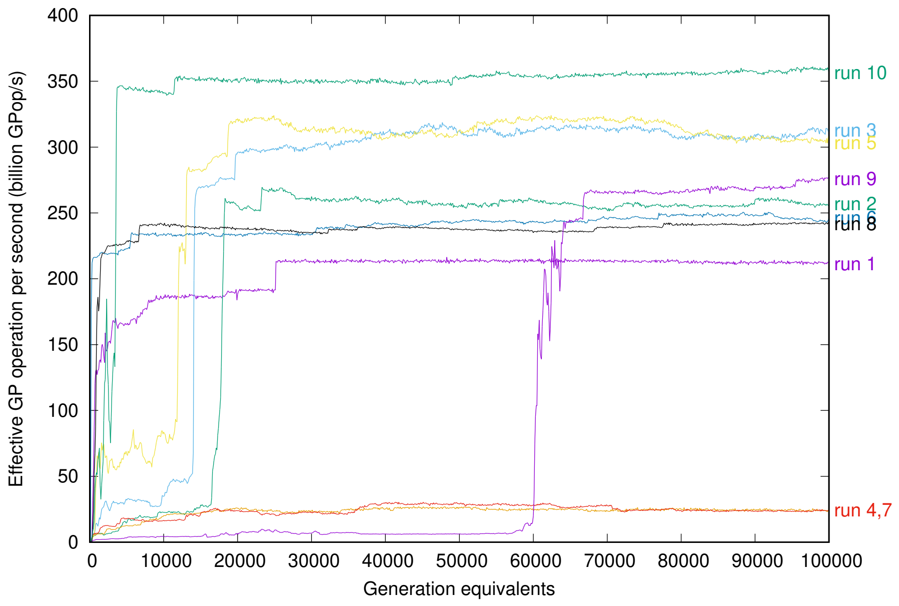

Fig. 1.7 Evolution of effective speed in ten runs of discrete Mackey-Glass chaotic sequence prediction with population of 500. Means every 100 generations. (Due to a transient, data for 19 Dec 3-4pm not plotted.)

1.5.4 Fitness Function

The first time `CalcFitness` is called, for efficiency, it extracts all the 1201 test inputs and their associated correct answers into arrays `Reg` and `Output`.

By default as many threads as the computer has cores will be used to calculate each new child's fitness. `nthreads` is set in `main` either by the user via the command line or otherwise to the default provided by the system routine `get_nprocs`. (Here we set `nthreads` = 19, as with AVX vector instructions each thread can simultaneously process 64 fitness cases and $19 \times 64 = 1216$, which exceeds our 1201 fitness cases.)

Each thread runs `interpret` which grabs and executes the next 64 fitness cases which are free (a mutex lock is used to ensure the threads do not try and run the same test cases). If need be, to balance the load between the threads, as a thread finishes each group of test cases it takes and processes the next free group, until all 1201 test cases are completed.

`interpret` calls `Interpret64`, which processes the 64 test cases in parallel. `Interpret64` starts by loading the given test cases into the eight registers (`a` to `h`, actually 64×8 are processed simultaneously), and then interpreting the `InstrLen2` effective instructions (held in array `Instr2`, Section 1.5.3). The program is evaluated on the 64 test cases simultaneously. At the end, the output register `a` is saved in array element `output` for those test cases (again 64 are processed simultaneously).

When all the threads have finished, `CalcFitness` compares the answer for each of the 1201 test cases with the correct one `Output[i]` and squares the difference (error_i). The individual's `fitness` is the sum of these squared errors. GPengine tries to minimise this error. Figure 1.1 plots falling Root Mean Squared error RMS $= \sqrt{\frac{1}{1201}\sum_{i=0}^{1201} \text{error}_i{}^2}$.

1.6 Results

Figure 1.1 shows typically even late into the run, linear GP continues to find ways to innovate. Also, not only do the programs increase in size (Figure 1.8), but so too does the number of instructions actually executed (Figure 1.9).

We anticipated the possibility of power law [Langdon, 2000] or even exponential [Nordin *et al.*, 1995] growth in program size. However only one run of ten shows almost continual rapid increase in program length (run 6 blue line Figure 1.8). The others show slower growth, often followed by a rapid increase phase. The impact of the size limit, 4 000 000, is clearly seen in Figure 1.8.

Figure 1.10 concentrates upon the period of explosive growth in program length. Figure 1.10 plots the average program size in the population from the last time it

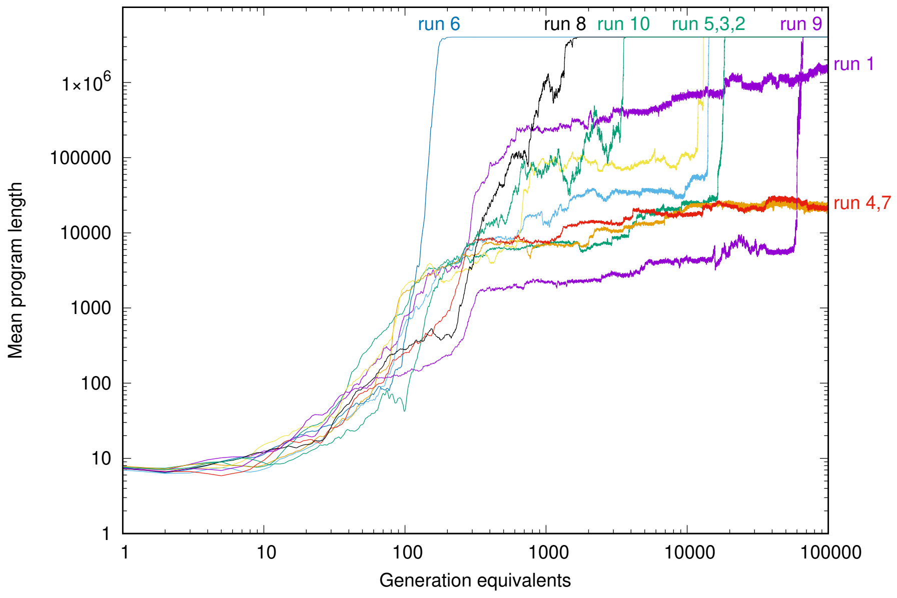

Fig. 1.8 Evolution of average genotype length in ten runs of discrete Mackey-Glass chaotic sequence prediction with population of 500.

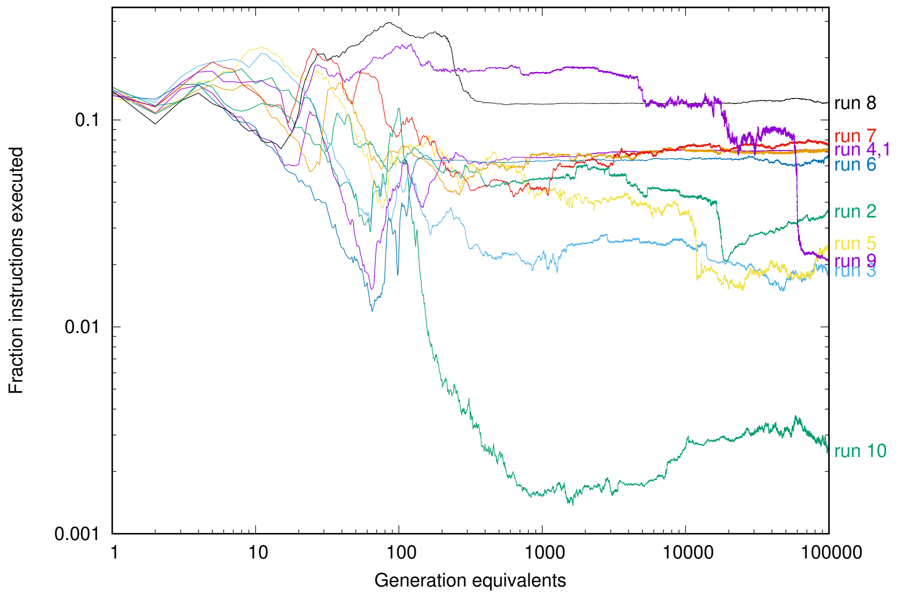

Fig. 1.9 Eliminating known introns so only a fraction of the genome is executed (saving 3.3–700, median 15.7).

exceeds 100 000 instructions until when many crossovers are impacted by the maximum program length (Section 1.5.1). So that exponential changes would appear as straight lines, rather than curves, Figure 1.10 uses a log vertical scale. If we consider only the end of the runs, 60% of them (runs 2, 3, 5, 6, 8, 10) appear to suffer exponential growth in size at some point [Banzhaf *et al.*, 1997].

Figure 1.11 shows, although innovation continues, the rate of fitness improvement appears to fall more-or-less linearly with increase in program size (see also Section 1.6.6, page 27). Notice although there is considerable noise and each run has a different slope, in every case the RMS best fit regression line approaches the origin. That is, the

$$\text{innovation rate} \propto \frac{1}{\text{program length}} \tag{1.1}$$

Even with relatively weak selection pressure [Goldberg, 1989] and steady state [Syswerda, 1990] with binary tournaments [Blickle, 1996; Langdon, 1998], we see evidence of population convergence. For example, Figure 1.12 shows the populations convergence in the sense that by the end of the runs typically more than 100 individuals have the best fitness. Whilst Figure 1.13 shows even stronger, in most runs (runs 2, 3, 6, 8, 9, 10) more than 40% of individuals return identical values across all 1201 training cases. That is, even though the population does not predict the Mackey-Glass sequence exactly, many individuals within the population make exactly the same mistakes as each other.

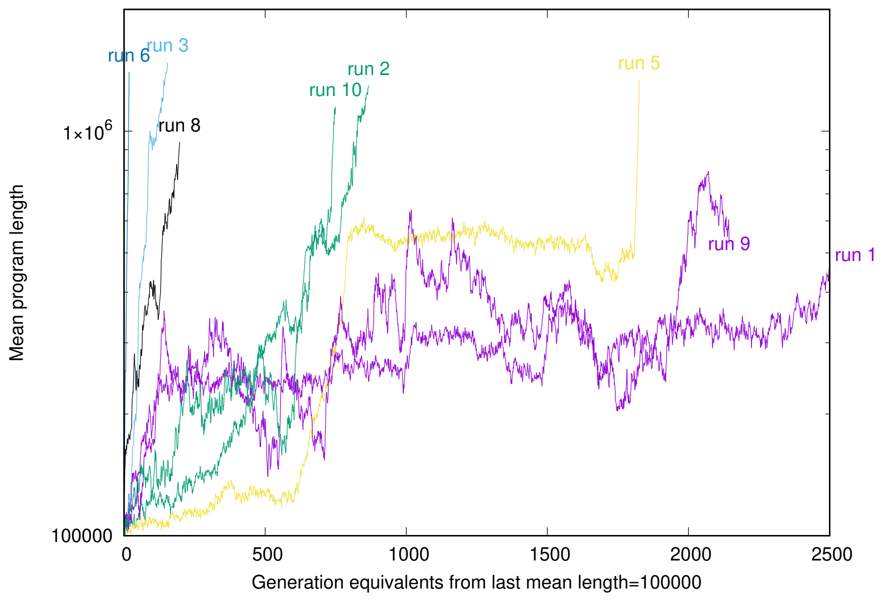

Fig. 1.10 On a log scale, mean genotype length after length exceeds 100 000 up to when maximum length (4 000 000) impedes growth. Note two runs never reached 100 000 but the ends of runs 2, 3, 5, 6, 8 and 10 seem to approximate exponential growth, albeit with different time constants.

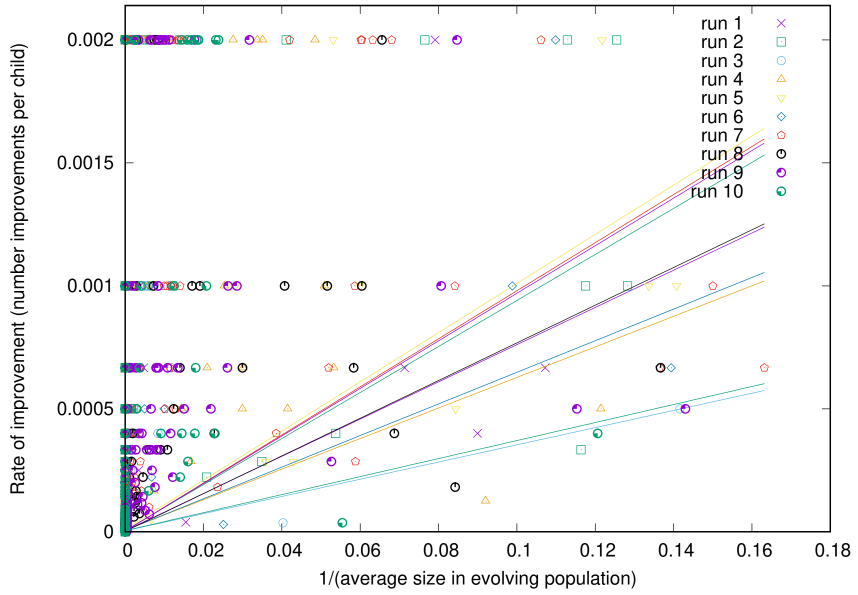

Fig. 1.11 Rate of improvement. Straight lines show linear RMS best fit.

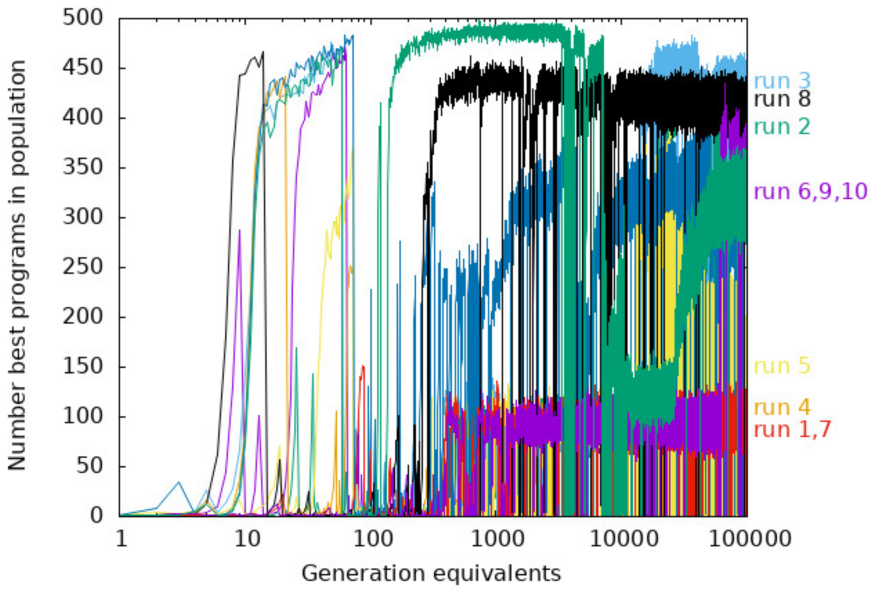

Fig. 1.12 Mackey-Glass linear GP population convergence. Number of programs with the best fitness. Log x-axis.

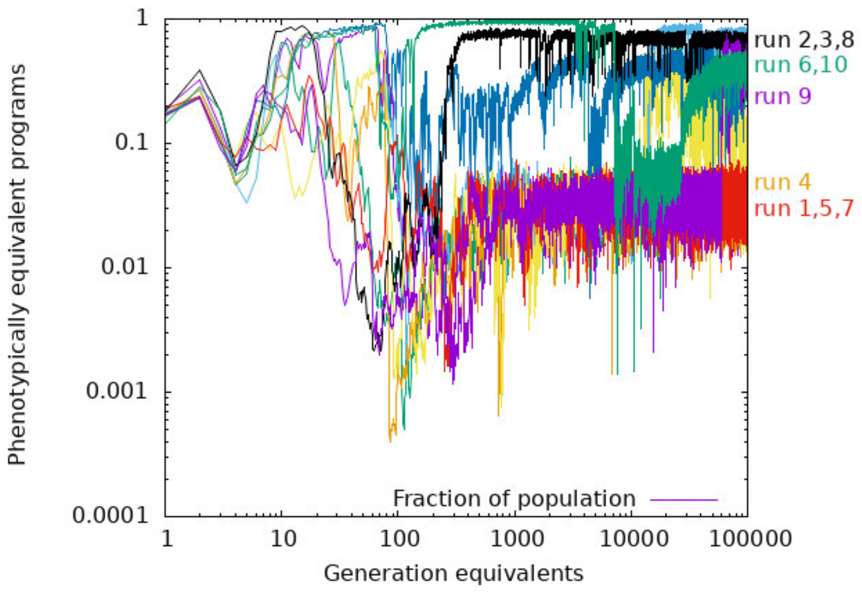

Fig. 1.13 Mackey-Glass linear GP population convergence. Fraction of population which calculate identical answers across 1201 test cases. Log scales.

1.6.1 Reminder of Entropy

Entropy is a property of distributions; entropy $= -\sum p \log p$. Where p is the probability of an event and the sum $\sum$ is over the whole probability distribution. In information theory it is common to express entropy in terms of bits and so log to the base 2 (i.e. $\log_2$) is used. Since p cannot exceed 1, $\log_2$ cannot exceed 0 and so the minus sign outside the summation ensures that entropy cannot be negative. Also if $p = 0$, $p \log p$ is taken as 0.

A rare event has a small chance of occurring, i.e. p is small, and therefore $-\log p$ is big. (For example, a one in 1024 chance event $p = 1/1024$ gives $-\log_2 p = 10$.) We can think of $-\log p$ as how surprising the event is and so we can think of entropy $= -\sum p \log p$ as how surprising on average the whole distribution is.

For a finite discrete distribution maximum entropy (maximum surprise) occurs when each possible event occurs with the same probability. If there are n events $p = 1/n$, then the entropy of the distribution is $= -\sum p \log_2 p = -n(1/n)\log_2(1/n)$ $= -\log_2(1/n) = \log_2 n$ bits.

If a lot of data, independently drawn at random from a distribution, is to be transmitted without error, the entropy of the distribution gives the maximum possible degree of compression. For example, a binary file might have an entropy of 8 bits per byte, whereas a text file of numbers (0-9 and space) might have entropy of $-11\frac{1}{11}\log_2\left(\frac{1}{11}\right) = log_2(11) = 3.46$ bits per character. Thus transferring the compressed text file can be done at best by transmitting 3.46 bits per character but the best compression for the binary file is 8.0 bits. Of course typical files are not random, nevertheless on large files compression tools, such as gzip, often approach compression ratios suggested by entropy.

1.6.1.1 Entropy: small distribution example

Suppose we have only two outcomes (e.g. 0 and 1), with probabilities p and q (note $p + q = 1$). Figure 1.14 plots the entropy $= -p\log_2 p - q\log_2 q$ of the distribution for different values of p. Notice when we are almost certain of the outcome (p or q is near 1.0), entropy is low. Conversely, when either 0 or 1 are similarly likely ($p \approx q \approx \frac{1}{2}$), we do not know which we will get (so either 0 or 1 will be a surprise) and entropy is near its maximum value, 1 bit.

1.6.1.2 Entropy: Mackey-Glass Benchmark distribution

If we look at the 1201 data points in our fitness set (Figure 1.2) we can see only one has the value 32 (Figure 1.3). So we can assign value 32 the probability $\frac{1}{1201}$. Whereas the value 145 occurs 30 times (probability $\frac{30}{1201}$). For small distributions we can take the summation ($\sum$) in our entropy calculation across the whole distribution. Here we know our data values all fit into a byte so summation over 0..255

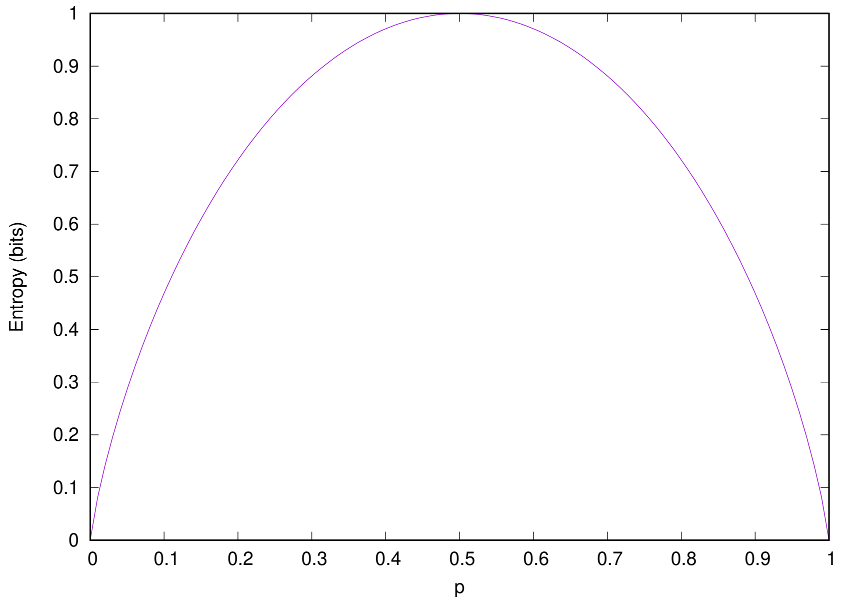

Fig. 1.14 Entropy expressed of distribution with only two possible outcomes, Section 1.6.1.1

($\sum_0^{255}$) is more than sufficient. (For larger ranges we use the C++ standard template library's hashmap routines.) Thus the entropy of discrete Mackey-Glass Benchmark distribution is $= -\sum_{i=0}^{255} p_i \log_2 p_i = 6.67$ bits.

Entropy is a very robust measure of distributions; in that, in many cases, "small" changes to a given distribution can be made which changes its entropy very little. Indeed in many cases the entropy of an actual distribution will be close to (but not exceed) that of a "Normal" or Gaussian distribution with the same standard deviation σ, whose entropy is $2.04 + \log_2 \sigma$ bits [Langdon and Clark, 2025, Appendix][2].

1.6.1.3 Entropy: Program State

In the following Sections (1.6.2–1.6.4) we use entropy to measure the information content within programs. (For our purposes we exclude external files and I/O devices which the program may read and/or write.) Although information can be stored in many ways, typically we need only consider the memory a program is using, e.g. control flags, registers, stacks and RAM. Sometimes memory struc-

[2] The entropy of a Gaussian distribution is $\frac{1}{2}\log\left(2\pi e\sigma^2\right) = \frac{1}{2}\log\left(2\pi e\right) + \log\sigma$. Notice a straight forward approximation is entropy = $\log_2$ (standard deviation) in bits. And that the constant term, $\frac{1}{2}\log\left(2\pi e\right)$, contains all our favourite mathematical constants ($\frac{1}{2}$, 2, π and e) and numerically, when using $\log_2$, evaluates to about two. Although the Mackey-Glass Benchmark distribution (page 4) is far from a smooth Gaussian, yet a Gaussian distribution with the same standard deviation (30.17) has an entropy of 6.97 bits, compared to the Mackey-Glass Benchmark's 6.67 bits.

tures (e.g. stacks) are dynamic. Typically deleted memory or memory past the stack pointer are viewed as inaccessible and ignored.

In our case, and in the case of many linear genetic programming systems, information is only stored in a fixed number of registers. (Here for simplicity we assume all registers are identical and they can all be both read and written to.)

For entropy calculations we have to treat the memory as a whole (not as 8 separate registers) and find the distribution of combined values it contains across all 1201 fitness cases. For example, at the start of the program, all the registers contain a copy of the input data, so all programs start with the same data in their registers and hence they all start with the same entropy, the entropy of the inputs $= -\sum_i p_i \log_2(p_i)$. Where the subscript i refers to the combined whole memory state (here 64 bits) that actually occurs in the computer when each program starts, and p_i is the fraction of the fitness cases where i occurs. I.e., $p_i = n_i/1201$, where n_i is the number of times i occurs when running all 1201 fitness cases. (As mentioned above) since i refers to 64 bits of state, we use the GNU STL hash table routines to count n_i and thus calculate entropy. Although there are 2^{64} possible states, there are only 1201 fitness cases, so even if each of the fitness cases were unique the entropy could only be at most $\log_2(1201) = 10.23$ bits (Due to repetitions the actual input entropy is 10.22 bits, see Figure 1.17, page 21.)

1.6.2 Generation 500, Entropy Loss

To avoid excessive run time, we switch to analysing in detail only the first 500 generations of our ten runs. In particular we look at the last 500 children created between generation 499 and 500, by which time they are already quite long and some of them are high scoring.

Figure 1.15 shows the entropy, of children created by crossover/mutation (i.e. excluding the 10% which are identical to their parents). To simplify, we only consider the first of the pair of children created by crossover/mutation, i.e. on average 90% of 250 = 225 for each of our ten runs. Figure 1.15 shows the internal entropy (i.e. the entropy of the eight registers) of the evolved programs as they execute across all 1201 test cases. Notice at the start of all programs, all 1201 test cases are loaded into the registers and so their entropy is the same as the entropy of the test cases, i.e. 10.22 bits. As the programs run they manipulate their registers, calculating new arithmetic values from the registers and the constants and overwriting registers with these new values. Notice that the programs are closed systems and so cannot invent new information as they execute. Thus if information is lost (by overwriting information rich registers), then it is lost for the rest of the program's execution. That is, entropy falls monotonically. In Figure 1.15 the numbers of instructions from the start of the program is plotted (on a log scale) horizontally. Figure 1.15 confirms entropy always falls monotonically. Notice at the end of a program's execution, if it is to predict the sequence well, it will need to have a good approximation of the sequence in the output register `a`, so we would expect the entropy of all the registers to be at least that of the output sequence, 6.67 bits.

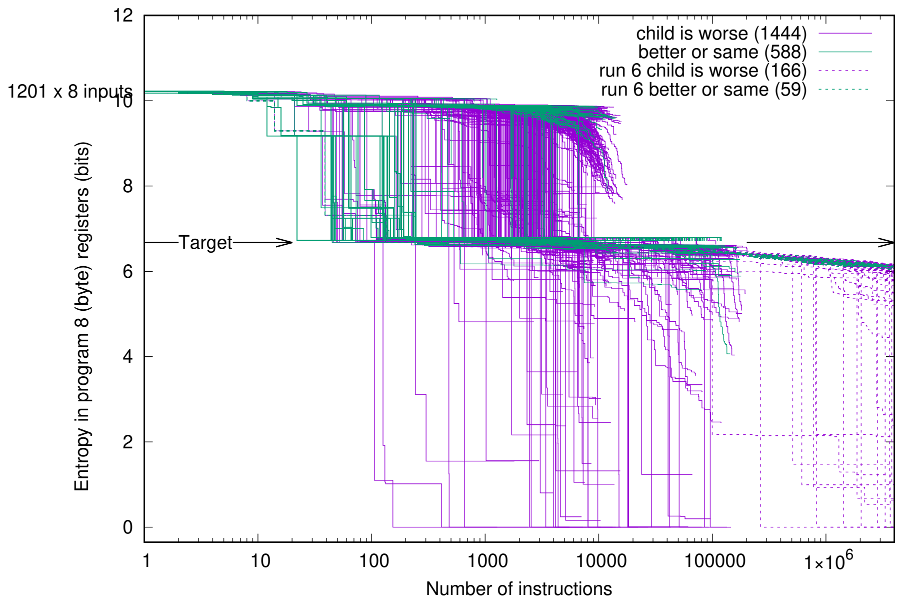

Fig. 1.15 Entropy within 1st child of fit parents at generation 500 during their execution (horizontal axis, log scale). Children who do worse than their 1st parent in purple. Run 6 parents are maximum length (4 10^6), they are plotted with dashed lines.

Figure 1.15 splits the programs into those with worse fitness than their parent (purple) and others (green, 205 of 646 are better than their first parent). Of those that are the same or better, almost all have a final entropy above or near that of the output sequence (6.67 bits). Indeed only two have a terminal entropy less than 5.67 bits.

In contrast 98 (of 1610, 6%) of those that are worse, have entropy below 5.67 bits. Indeed 29 terminate having calculated a constant (0 entropy).

Of the 299 best programs (i.e. with an RMS error ≤ 2.5), 210 terminate with an entropy between 6.49 and 6.76. Notice when these 210 good programs finish their seven non-output registers contain almost no additional information. That is, typically good highly evolved programs succeed by reducing their whole entropy from that of the input pattern to be close to that of the desired target pattern.

Although Figure 1.15 hints that low entropy might be used as a way to detect, and so abort early, children with low residual entropy, here it suggests the saving in runtime (even if entropy calculation was free) might not exceed 5%. [Maxwell III, 1994] proposed during fitness evaluation allotting each child a small unit of run time resources. The execution of those doing poorly would be halted. Whilst those doing well enough that they might win a tournament, would be given additional computer time and allowed to continue. Notice Sid Maxwell assumed partial fitness could be calculated as the program ran (anytime fitness [Teller, 1994]), whereas here we can look inside the program as it runs and calculate the entropy remaining within it.

Rather than seeking a way of using entropy to optimise runtime of an existing system, a suitable compromise might be to explicitly make entropy part of the fitness function. For example, we might explicitly say we will measure internal entropy after 100 instructions and reject any program with entropy below 90% of the target's entropy, without running it to completion. Alternatively, and this is more speculative, perhaps there is a way of estimating which mutations, or even crossovers, will destroy information, without evaluating the whole program. For performance, we might be prepared to accept that this will be approximate and will introduce noise, and that this will change in detail the course of individual runs.

1.6.3 Information Loss and Preservation

Figure 1.16 concentrates on a particular fit program produced by crossover and mutation at generation 500 in run ten. Notice the tiny fraction of instructions (shown with non-vertical arrows) which cause entropy loss and the rapid resynchronisation of the child's execution with that of its parents following disruption by four point mutations and the two crossover boundaries.

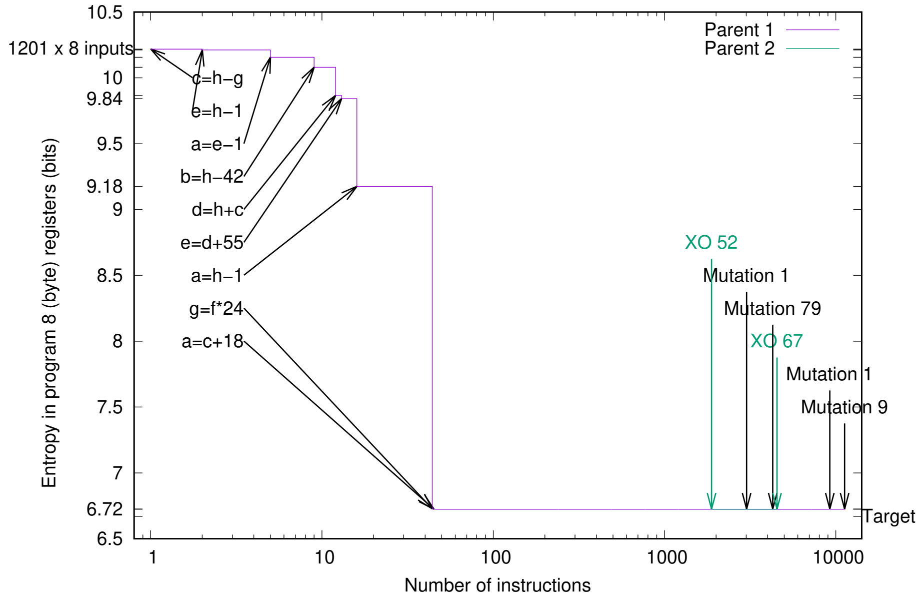

Fig. 1.16 Entropy within 1st example fit program at generation 500 of run 10 during its execution (horizontal axis, log scale). Crossover fragment (XO) from 2nd parent in green. Similarly location of four point mutations shown with vertical arrows. Numbers following XO and Mutation are the number of instructions which are disrupted before the contents of all the registers return to their values in the parents. Left hand side arrows highlight all the instructions which cause information loss on the 1201 test cases.

The first instruction c=h−g calculates the difference between registers h and g and saves it in register c, destroying the contents of c. In terms of predicting the next value in the Mackey-Glass sequence, this instruction calculates the signal's gradient between 2 and 1 time steps ago and saves it in the register containing data from 32 steps ago. Notice the loss of data from 32 steps ago does not seriously impact the program's ability to predict the Mackey-Glass sequence (Figures 1.2 and 1.4). Indeed it gives only a small reduction in the entropy in all eight registers. Also we would expect the gradient to be very useful in predicting the next value. Indeed it is one of the 34 instructions required to calculate the program's output (The subtraction h−g is used 212 times in the program's 11 326 instructions [Langdon and Banzhaf, 2005]. In a randomly created program of 11 326 instructions we would anticipate h−g occurring 8.8 times.)

The next instruction, e=h−1, also looses entropy but it does not influence the program's output.

The third instruction a=a+1 again does not influence the program's output. Also by incrementing register a's value it does not loose information. If we know a has been incremented, we can recover its previous value by decrementing it. I.e., unusually for an arithmetic operation, increment is a reversible instruction.

Because of the previous instructions the fourth instruction, h=h+c, also does not loose information. Remember c = h−g, so now h = $2h_0-g_0$. That is, the new values of h,c contain no more (or less) information than they did before h was over written by h+c. In terms of predicting the value at the next time step, it may be that doubling the weighting of the most recent value (h_0 1 time step ago) compared to that of the previous (g_0 2 time steps ago) makes sense. The fourth instruction, h=h+c, does influence the program's final output and again is repeated many (291) times.

As shown in Figure 1.16, the fifth instruction a=e−1 does reduce entropy (by overwriting register a) but again it does not influence register a's final value.

Figure 1.16 shows only nine instructions destroy information and that by instruction 45 the program has reduced its internal information to 6.72 bits, which is almost that of the target sequence (6.67 bits). Amazingly the remaining thousands of instructions simply move information between the registers and do not loose any more. Secondly, the child program quickly recovers its parents' behaviour after both crossover and multiple mutations.

1.6.4 Information Preservation

We randomly selected ten of the thousands of instructions in the example fit program shown in Figure 1.16. All ten preserve entropy. Figure 1.18 shows the distribution of memory values before and after eight of the ten. (Figure 1.20 is typical of the other two.) For comparison Figure 1.17 shows the initial distribution before any program runs.

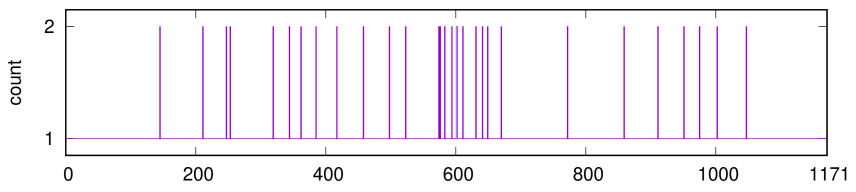

Fig. 1.17 Initial distribution of memory values before a program starts. Note 30 of the 1201 test cases give rise to repeated memory patterns in the 8 registers, so reducing entropy from that of a uniform distribution ($\log_2 1201 = 10.23$) to 10.22 bits.

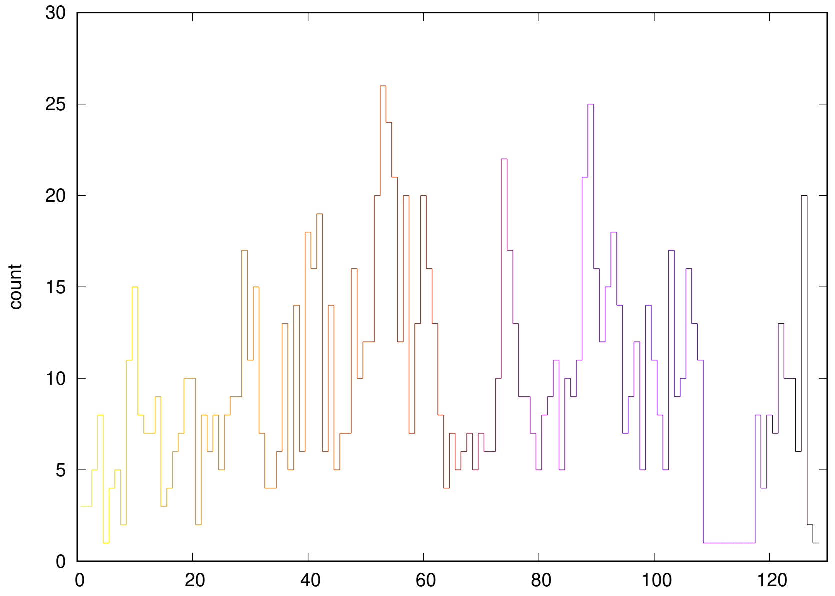

Fig. 1.18 Distribution of memory values before and after 8 randomly selected instructions (200, 2477, 3225, 4037, 4509, 4576, 7243 and 8784). Like all but 9 instructions in this program (Figure 1.16) they do not change entropy (i.e. it remains 6.72 bits). Notice although there are 1201 test cases, they set only 128 different combined memory values across all eight registers.

The first thing to say is that although these instructions change data values in the program's memory, remarkably the distribution of values is identical before and after these eight instructions (i.e. instructions 200, 2477, 3225, 4037, 4509, 4576, 7243 and 8784, Figure 1.18). Since the distributions are unchanged, of course the entropy is unchanged.

The key to understanding why irreversible operations can be isentropic and not loose entropy, is to consider the distribution of memory across the 1201 test cases. We can use the memory contents to partition the test cases. For example, when the program starts some test cases have identical inputs and so initially give rise to identical memory contents (Figure 1.17). That is, the program starts with 1171, rather than 1201, partitions, If two or more test cases give rise to the same memory contents, the next instruction will give rise to new contents but (since the programs are deterministic) the two test cases will both have the same new memory contents and so remain together in the same partition. In fact they will remain together until the program terminates[3]. If the partition of all the test cases is the same, the distribution and so its entropy remains the same. Thus although not reversible the instruction does not loose entropy.

Of course there are cases where entropy is lost. Suppose two test cases which have different memory contents (i.e. are in different partitions) execute an instruction which although it has different inputs gives the same output. This will cause the partitions holding the two test cases to merge. Thus changing the overall distribution of partitions and reducing its entropy[4]. We can see this by working through an example. (The next section (1.6.5) considers how evolved programs can often continue to pass information through seemingly arbitrary code changes.)

Consider in detail the first randomly chosen instruction, `a=g/42` (Figure 1.19). It overwrites register `a` (original values between 0 and 252) with small integer values generated by dividing the values in register `g` (values between 24 and 255) by 42 (new values between 0 and 6). If we consider only register `a` we would expect integer division by 42 to give a low entropy distribution. Indeed the operation of replacing `a`'s current value by values between 0 and 6 (given by `g/42`) does indeed reduce the entropy held by `a`. However, although it does change locally, it is important to consider the sparse data pattern held in the rest of the program.

Almost all instructions in the program do not change entropy. If we consider the program after instruction 45, the internal entropy is always 6.72 bits (Figure 1.16) and the contents of the registers partitions the 1201 test cases in the same way. Although after each instruction typically one register has changed, there is always an indirect one-to-one mapping via the other seven registers between the new register values and the old. For example (starting at the left in Figure 1.18, yellow), since 66/42 = 1, the register pattern (`a` to `h`) 156,2,136,1,0,92,$\underline{66}$,67 (which occurs three times) becomes 1,2,136,1,0,92,$\underline{66}$,67. Thus, the three new values of register `a` are 1 (which replaces 156). Of course the new pattern also occurs three times. Although 35 of the 42 values that register `g` could have had, which also set register `a` to 1 are present (values g=42..83 horizontal row of $\times$ at `a` (after)=1 in left part Figure 1.19), apart from `g`=66 they occur with different values in the other six registers. That is, they contribute to different partitions of the memory value distribution. The same holds across the whole of the 1201 test values. Thus, although values in the memory (here register `a`) have changed, the shape of the distribution and hence the entropy does not change either.

[3] In deterministic programs, if different test cases give rise to identical memory, then from that point on the execution of the test cases will remain synchronised (i.e., with identical memory contents) and so they will give identical outputs. Thus Figure 1.17 says we need only run 1171 of out 1201 test cases. Whilst Figure 1.18 says, we need only run this program on 128 test cases.

[4] Notice reversible instructions cannot change entropy. Suppose a reversible instruction's inputs on two test cases are different (i.e. they belong to different memory partitions). As reversible instructions are one-to-one, its outputs must also be different. Therefor it cannot cause the partitions to merge, so their distribution, and hence its entropy, will not change.

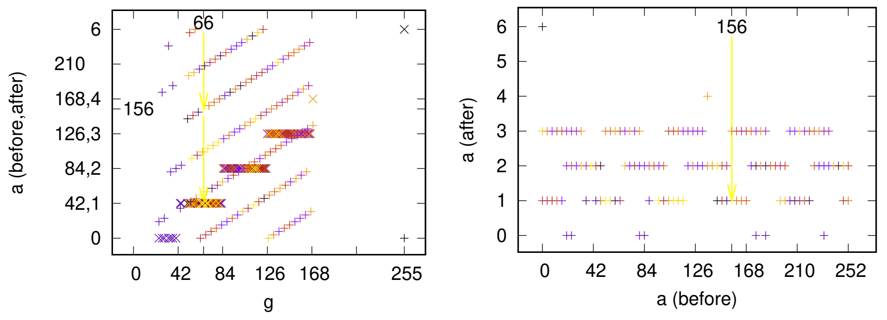

Fig. 1.19 Distribution of values in registers a and g before and after randomly chosen instruction 200 `a=g/42` in fit program at generation 500. Left: + show even before register a is overwritten × there is a structured relationship between a and g. Right: register a before and after `a=g/42`. Colour indicates position in left-right histogram sequence in Figure 1.18. Entropy (6.72 bits) is not changed by `a=g/42`.

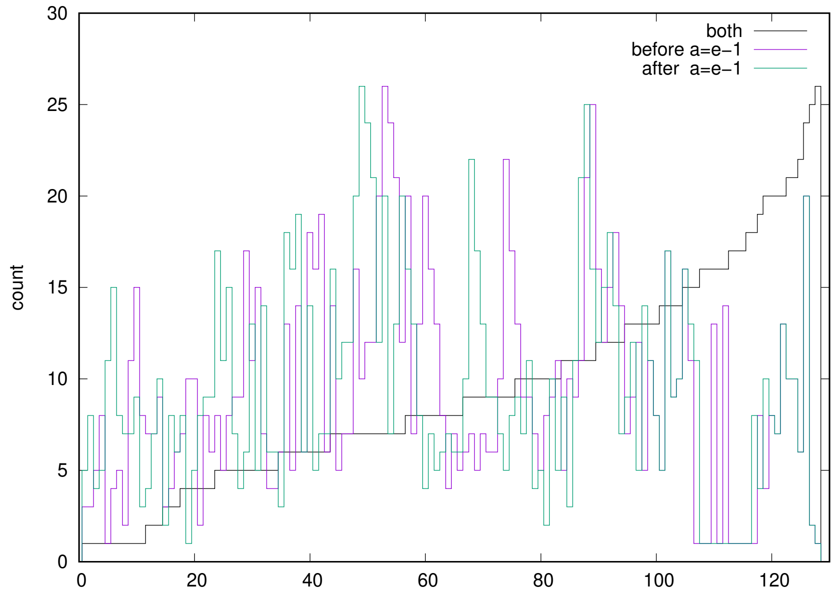

Fig. 1.20 Distribution of memory values before (purple) and after (green) randomly selected instruction (887, `a=e-1`) overwrites a register. As Figure 1.18, although there are 1201 test cases, they set only 128 different memory values. The before and after distributions are essentially the same but variably shifted. This is confirmed by normalising, to give the same monotonically increasing curve for both (black), confirming entropy is not changed (6.72 bits).

Figure 1.20 shows that the situation is similar for the other two randomly selected points (877, 6120). Again entropy is still preserved. Figure 1.20 shows the distribution of memory values before the randomly chosen instruction (877) overwrites a register and afterwards. (The plot for instruction 6120 is similar.) In both these examples, the memory values distribution after the instruction is shifted wrt to before. However, in both Figures 1.18 and 1.20 essentially the x-axis is given by the somewhat arbitrary sorting order provided by the hashmap routines. The shifting of the distribution of memory values along the x-axis makes no difference to its entropy. To emphasise this Figure 1.20 shows that if we normalise by sorting the before and the after histograms, they both give the same histogram (plotted in black). Confirming that although the instruction overwriting a register changes memory values, in this fit evolved program, it does not in fact change the program's internal information contents.

1.6.5 Evolved Code Robustness

Referring back to Figure 1.16, we see each of the genetic changes (two crossover points and four point mutations) all cause disruption but none of them cause the program's internal entropy (6.72 bits) to change. This is somewhat at odds with our general notion that in human written programs, mutations and other forms of disruption fail to make an external impact (Failed Disruption Propagation) because they are followed by entropy loss which dissipates their disruptive effect before it can reach an output [Petke *et al.*, 2021]. Nonetheless the following sections discuss how the evolved code removes the disruption.

The number of test cases where any part of the memory has a disrupted value falls monotonically as program execution continues past the disruption point (as it should, not plotted). Figure 1.21 shows disruption in terms of number of registers and number of test cases. Since a perturbation to one register can spread to more, the trace need not be monotonic. If the program were fragile and the disruption large, the disruption could spread to include all the registers and never die away (as happened sometimes to other evolved but unfit programs). Figure 1.21 is plotted on a log scale to test the idea that the fall in this measure of disruption might be exponential (a straight line on log scale), or at least approximately exponential.

In two cases ($+$ and $\times$) disruption is lost immediately. In one mutation (black) and the first crossover (XO 1), the fall might be (generously) regarded as a noisy exponential. In the other cases, there are multiple large rises and falls and so a simple exponential seems a poor model of the actual behaviour.

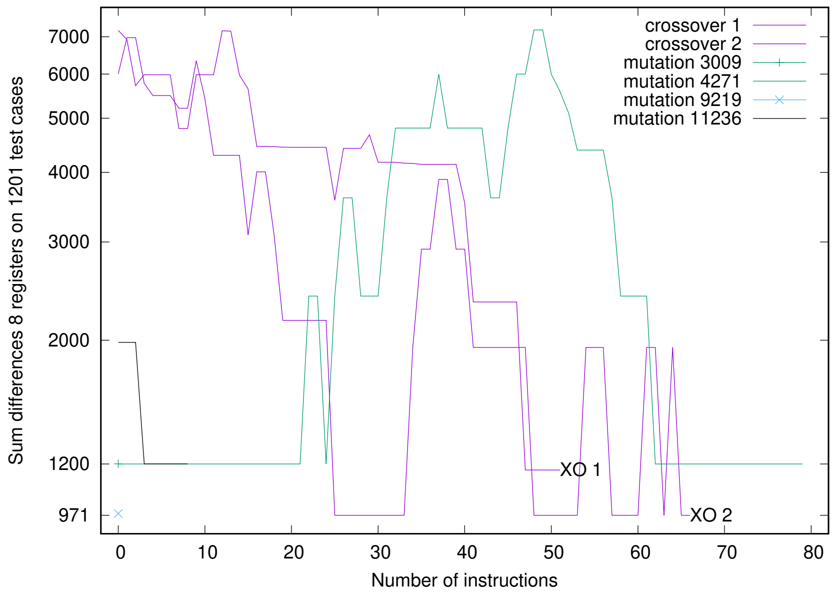

Fig. 1.21 Distribution in terms of number of registers and number of test cases following crossover (purple) and point mutation sites versus number of unperturbed program instructions before disruption falls to zero. 1^{st} example fit program at generation 500 of run 10. Vertical axis on log scale.

1.6.5.1 1^{st} Crossover point 1876, Disruption length 52

The child is first created by crossover and then subjected to four distinct point mutations, It can be thought of as a copy of its first parent with a code section (length 160) removed and replaced by randomly selected but fit evolved code from its second parent (length 2648), increasing its size by 2488 instructions. Naturally the introduced code disrupts its operation but as Figure 1.16 shows, this does not change its internal entropy.

At the start of the crossover section (purple XO 1 line in Figure 1.21) all 1201 test cases are disrupted (on average 5.98 registers per test case). It appears on at least one test case all registers, except `h`, initially contain different values in the child than they had when the code was executed in the second parent. (Remember for predicting the next value, the previous value, initially held in register `h`, is perhaps the most useful data item. So it makes sense for evolution to protect `h`.) It seems reasonable to assume that each of the registers with a different value would have to be overwritten at least once before their contents resynchronise. During the 52 instructions where the contents differ registers `a` and `b` are written to 21 and 8 times but `d` and `f` only 4, `c` 3 and `g` only 2, both of which the new value of `g` does not depend on the other registers. Indeed the instruction `g=e-1` which synchronised the child with execution in its second parent is the first time register `g` is overwritten with a value that depends on the others.

1.6.5.2 Mutation 3009, Disruption length 1

The point mutation of instruction 3009 (+ in Figure 1.21) replaces `a=h-1` (in the crossover region taken from the 2nd parent) with `a=b-1`, typically changing the value in register `a`. However the next instruction (`a=f-1`) also sets register `a` so overwriting the mutated value in `a`. Thus the disruption lasts only one instruction.

1.6.5.3 Mutation 4271, Disruption length 79

The second point mutation replaces an addition with a protected division. I.e. `c=h+c` becomes `c=h/c` (green line in Figure 1.21). This changes the contents of register `c` on all but one test case. The disruption remains in register `c` until 22 instructions later when `e=c+b` spreads it to register `e`. However the disruption to `e` is short lived, as two instructions later it is overwritten by `e=a+1`, so restoring its unmutated value. But then `c`'s disruption is spread to `d` (by `d=h+c`) and then in turn to `f` (by `f=d+55`). Shortly later `d` is restored to its unmutated value (by `d=b-1`) and then reinfected three instructions later (by `d=c-105`). The disruption is spread further in the next instruction (`e=c-b`) reaching a new peak five instruction later to include also `b` (by `b=d/g`). However the next instruction (`c=h+7`) resets `c`, followed by resetting `f` (`f=h-110`). From this point, disruption climbs steadily (via `a=b/7`, `f=d*55`, `e=d*74` and `g=e-1`). Until at the peak, all but two registers (`h` and `c`) have been influenced by the mutation. From the peak the influence of the mutation falls steadily to nothing (via `b=h-1`, `a=h-116`, `e=h/1`, `d=b+88`, `f=d+55` and `g=e-1`).

It is tempting to think of this as a random walk or a Gambler's ruin [Langdon, 2002] but remember these are not random instructions, they have been honed by evolution for 500 generations. Also remember none of them loose information and (as with all the instructions in these section) entropy is preserved throughout (Figure 1.16).

1.6.5.4 2nd Crossover point 4524, Disruption length 67

This section shows that the disruption caused by swapping to code from the other parent (Section 1.6.5.1) is similar to when (at instruction 4524) execution returns to the first parent.

Again all 1201 test cases are disrupted, with on average one fewer registers ($\approx$5.00) disrupted (purple line starting at 6002 on left hand side of Figure 1.21). Again it seems register `h` is not disrupted. And this time also the output register `a` also takes the same values in the child as it took in the first parent. So, it appears, that transition of execution across the junction between second parent and first parent code does not disturb these two registers.

As expected, the distribution of registers over write is similar to that at the start of the crossover fragment. Again it seems reasonable to assume the disruption continues until all the initially different registers are overwritten at least once (or perhaps twice). This time it takes slightly longer (67 instructions rather than 52). The instruction which finally removes all the disruption is `f=h/1` which sets `f` to `h`. Notice, for each test case, `h` has the same value in both parents.

1.6.5.5 Mutation 9212, Disruption length 1

This point mutation replaces `a=g/g` with `a=a/g` typically changing the value in register `a` ($\times$ in Figure 1.21). However, like the point mutation at 3009 (Section 1.6.5.2), the next instruction also sets register `a`, meaning the disruption lasts only one instruction.

1.6.5.6 Mutation 11236, Disruption length 9

Mutation 11236 replaces `e=d/74` by `d=d/74` (black line in Figure 1.21). It is different from the others we have seen because it changes the output register. Therefore it impacts the new output register (`d`) and no longer updates the old output register (`e`). However `e` is reset three instructions later (by `e=h/1`). Similarly the new value of `d` remains in it until nine instructions later when it is overwritten (by `d=b-1`) which resets `d` to its unmutated value. Hence although the mutation changes memory on all 1201 test cases, its influences only lasts 9 instructions.

1.6.6 Rate of Improving Evolved Code Slows with Program Length

In the previous sections we have shown that programs evolve long robust tails. We suggest that regions of the program with high information (high entropy), which are near the program's start, will be more fragile and so will be liable both to disruption and improvement. Figure 1.22 concentrates upon our ten runs at generation 500 and shows all but 15 of 1074 programs were disrupted when the first genetic change was in a region where their parent's entropy was more than 8 bits. (I.e., almost all children had a different fitness to that of the parent they inherit their first instruction from, if some of the genetic differences lay in information rich code.) In contrast when all the genetic changes were in regions with entropy below 8 bits (typically 6.8 or less), more than a 36% (427/1183) of children had the same fitness as their parent.

Notice this provides a plausible argument to explain why the rate of innovation falls with program length (Figure 1.11). If we assume for each run the populations converge so that near the start of each program there is a fragile high entropy region of more or less fixed size (Figure 1.15). Then the chance of either mutations or crossover points landing in it falls in proportion to how much of the whole program it occupies. If we assume only changes to the information rich code lead to fitness

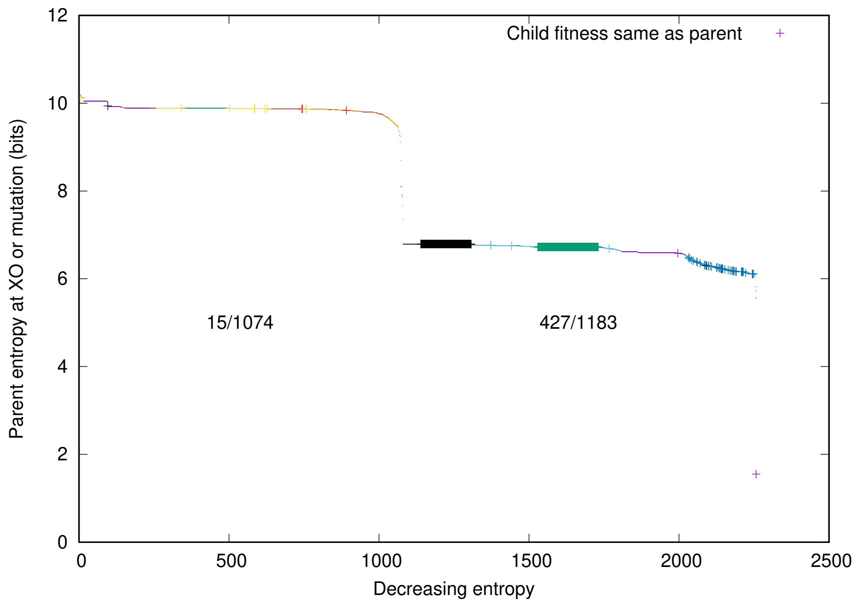

Fig. 1.22 Impact of crossover/mutation on 1st child of fit parents at generation 500. + child's fitness is the same as 1st parent. Better or worse both plotted with dot. Vertical axis shows highest entropy within parent at the crossover and mutations points used to create the child. Horizontal axis 2257 children from ten runs sorted by decreasing entropy (y-axis).

improvements, then this leads immediately to Equation 1.1 (page 12). That is, the rate of innovation falls inversely with increasing program length.

This suggests prolonged evolution of linear GP is both like tree GP and different. It is like tree GP, in that the rate of improvements also falls in proportion to programs size. However different in that in simple tree GP, populations of evolved trees converge around the trees root nodes [Langdon, 2022a]. Note that the sensitive part of tree genetic programming is near the program's output whereas in linear GP its near the programs's inputs.

1.7 Conclusions

Even with modest hardware, long term evolution experiments (LTEE) are possible with a simple linear genetic programming system (GPengine) in a few weeks, rather than 38 years. Nevertheless bloated populations of long programs, like those of deep trees, suffer from convergence and become very robust, causing the rate of innovation to reduce in proportion to the programs' increase in size.

Information theoretic analysis shows to predict the chaotic time series, fit evolved programs rapidly loose the information given by their inputs in order to approach that of the desired output. However once the entropy of the target sequence has been (approximately) reached, they become enormously stable, retaining exactly the same entropy despite executing thousands of irreversible instructions.

Detailed analysis shows that such programs are often resilient to both crossover and mutation. Typically fit programs restore their whole internal state (and therefore their final outputs) by executing no more than a few dozen instructions after the disruption caused by the mutation or at either of the transition points between code inherited from the child's two parents.

Potentially modest runtime savings could be obtained by cutting short the execution of programs which have lost too much entropy, since we know before they terminate that they cannot possibly approximate their target well and so must have a poor fitness. Also information theory using internal entropy (here obtained at run time, but potentially estimated via program analysis) could be used to predict program performance.

The intention is to continue to enhance GPengine and use it as a framework to support analyse of open-ended co-evolution of multiple data sharing learning programs. So far we have used our experience with AVX vector instructions [Langdon, 2020a; Langdon, 2022c] and pthreads on a multi-core computer to speed up GPengine's interpreter. Based on previous experience we hope that use of GPUs [Langdon and Banzhaf, 2008; Langdon, 2010; Langdon *et al.*, 2014; Langdon *et al.*, 2017] will also greatly increase performance, allowing continual learning theory and experimentation on relatively modest hardware.

Code based on Peter Nordin's GPengine and the discretised Mackey-Glass dataset are available via `https://github.com/wblangdon/GPengine`.

Acknowledgement

I would like to thank Sara Silva, John Woodward, Eric Medvet, Wolfgang Banzhaf, Leonardo Trujillo, Laurie Tratt and my anonymous reviewers.

References

Altenberg, 1994. Lee Altenberg. The evolution of evolvability in genetic programming. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 3, pages 47–74. MIT Press, 1994.

Angeline, 1994. Peter John Angeline. Genetic programming and emergent intelligence. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 4, pages 75–98. MIT Press, 1994.

Banzhaf and Langdon, 2002. W. Banzhaf and W. B. Langdon. Some considerations on the reason for bloat. *Genetic Programming and Evolvable Machines*, 3(1):81–91, March 2002.

Banzhaf *et al.*, 1997. Wolfgang Banzhaf, Peter Nordin, and Frank D. Francone. Why introns in genetic programming grow exponentially. Position paper at the Workshop on Exploring Noncoding Segments and Genetics-based Encodings at ICGA-97, 21 July 1997.

Banzhaf *et al.*, 1998. Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, San Francisco, CA, USA, January 1998.

Blickle, 1996. Tobias Blickle. *Theory of Evolutionary Algorithms and Application to System Synthesis*. PhD thesis, Swiss Federal Institute of Technology, Zurich, Switzerland, November 1996.

Brameier and Banzhaf, 2007. Markus Brameier and Wolfgang Banzhaf. *Linear Genetic Programming*. Number XVI in Genetic and Evolutionary Computation. Springer, 2007.

Chihoub *et al.*, 2025. Doha Chihoub, Coralie Pintard, Richard E. Lenski, Olivier Tenaillon, and Alejandro Couce. The evolution of robustness and fragility during long-term bacterial adaptation. *Proceedings of the National Academy of Sciences*, 122(16):e2501901122, 2025.

Francone *et al.*, 1999. Frank D. Francone, Markus Conrads, Wolfgang Banzhaf, and Peter Nordin. Homologous crossover in genetic programming. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1021–1026, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

Francone, 2001. Frank D. Francone. *Discipulus Owner's Manual*. 11757 W. Ken Caryl Avenue F, PBM 512, Littleton, Colorado, 80127-3719, USA, version 3.0 draft edition, 2001.

Goldberg, 1989. David E. Goldberg. *Genetic Algorithms in Search Optimization and Machine Learning*. Addison-Wesley, 1989.

Hansen, 2003. James V. Hansen. Genetic programming experiments with standard and homologous crossover methods. *Genetic Programming and Evolvable Machines*, 4(1):53–66, March 2003.

Koza, 1992. John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

Langdon and Banzhaf, 2005. William B. Langdon and Wolfgang Banzhaf. Repeated sequences in linear genetic programming genomes. *Complex Systems*, 15(4):285–306, 2005.

Langdon and Banzhaf, 2008. William B. Langdon and Wolfgang Banzhaf. A SIMD interpreter for genetic programming on GPU graphics cards. In Michael O'Neill, Leonardo Vanneschi, Steven Gustafson, Anna Isabel Esparcia Alcazar, Ivanoe De Falco, Antonio Della Cioppa, and Ernesto Tarantino, editors, *Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008*, volume 4971 of *Lecture Notes in Computer Science*, pages 73–85, Naples, 26-28 March 2008. Springer.

Langdon and Banzhaf, 2022. William B. Langdon and Wolfgang Banzhaf. Long-term evolution experiment with genetic programming. *Artificial Life*, 28(2):173–204, Summer 2022. Invited submission to Artificial Life Journal special issue of the ALIFE'19 conference.

Langdon and Clark, 2024. William B. Langdon and David Clark. Deep mutations have little impact. In Gabin An, Aymeric Blot, Vesna Nowack, Oliver Krauss, and Justyna Petke, editors, *13th International Workshop on Genetic Improvement @ICSE 2024*, pages 1–8, Lisbon, 16 April 2024. ACM. Best paper.

Langdon and Clark, 2025. W. B. Langdon and David Clark. Deep imperative mutations have less impact. *Automated Software Engineering*, 32:article number 6, 2025.

Langdon and Hanna, 2026. William B. Langdon and Carol Hanna. Improving a parallel C++ Intel SSE SIMD linear genetic programming interpreter. In Aymeric Blot and Oliver Krauss, editors, *15th International Workshop on Genetic Improvement @ICSE 2026*, Rio de Janeiro, 13 April 2026.

Langdon and Hulme, 2024. W. B. Langdon and Daniel Hulme. Sustaining evolution for shallow embodied intelligence. In Arsen Abdulali, Josie Hughes, and Fumiya Iida, editors, *Proceedings of 2023 International Conference on Embodied Intelligence, EI-2023*, volume 1321 of *IOP Conf. Series*, page 012007, Internet, Cambridge, UK, 20-22 March 2024. IOP Publishing. In EI 2024 but for production reasons published in EI 2023 proceedings.

Langdon and Nordin, 2001. William B. Langdon and Peter Nordin. Evolving hand-eye coordination for a humanoid robot with machine code genetic programming. In Julian F. Miller, Marco Tomassini, Pier Luca Lanzi, Conor Ryan, Andrea G. B. Tettamanzi, and William B. Langdon, editors, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 313–324, Lake Como, Italy, 18-20 April 2001. Springer-Verlag.

Langdon and Petke, 2015. William B. Langdon and Justyna Petke. Software is not fragile. In Pierre Parrend, Paul Bourgine, and Pierre Collet, editors, *Complex Systems Digital Campus E-conference, CS-DC'15*, Proceedings in Complexity, pages 203–211. Springer, September 30-October 1 2015. Invited talk.

Langdon and Poli, 1997. W. B. Langdon and R. Poli. Fitness causes bloat. In P. K. Chawdhry, R. Roy, and R. K. Pant, editors, *Soft Computing in Engineering Design and Manufacturing*, pages 13–22. Springer-Verlag London, 23-27 June 1997.

Langdon *et al.*, 2014. William B. Langdon, Marc Modat, Justyna Petke, and Mark Harman. Improving 3D medical image registration CUDA software with genetic programming. In Christian Igel, Dirk V. Arnold, Christian Gagne, Elena Popovici, Anne Auger, Jaume Bacardit, Dimo Brockhoff, Stefano Cagnoni, Kalyanmoy Deb, Benjamin Doerr, James Foster, Tobias Glasmachers, Emma Hart, Malcolm I. Heywood, Hitoshi Iba, Christian Jacob, Thomas Jansen, Yaochu Jin, Marouane Kessentini, Joshua D. Knowles, William B. Langdon, Pedro Larranaga, Sean Luke, Gabriel Luque, John A. W. McCall, Marco A. Montes de Oca, Alison Motsinger-Reif, Yew Soon Ong, Michael Palmer, Konstantinos E. Parsopoulos, Guenther Raidl, Sebastian Risi, Guenther Ruhe, Tom Schaul, Thomas Schmickl, Bernhard Sendhoff, Kenneth O. Stanley, Thomas Stuetzle, Dirk Thierens, Julian Togelius, Carsten Witt, and Christine Zarges, editors, *GECCO '14: Proceeding of the sixteenth annual conference on genetic and evolutionary computation conference*, pages 951–958, Vancouver, BC, Canada, 12-15 July 2014. ACM.

Langdon *et al.*, 2017. William B. Langdon, Brian Yee Hong Lam, Marc Modat, Justyna Petke, and Mark Harman. Genetic improvement of GPU software. *Genetic Programming and Evolvable Machines*, 18(1):5–44, March 2017.

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

Langdon, 1999. W. B. Langdon. Size fair and homologous tree genetic programming crossovers. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1092–1097, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

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

Langdon, 2002. W. B. Langdon. How many good programs are there? How long are they? In Kenneth A. De Jong, Riccardo Poli, and Jonathan E. Rowe, editors, *Foundations of Genetic Algorithms VII*, pages 183–202, Torremolinos, Spain, 4-6 September 2002. Morgan Kaufmann. Published 2003.

Langdon, 2010. W. B. Langdon. A many threaded CUDA interpreter for genetic programming. In Anna Isabel Esparcia-Alcazar, Aniko Ekart, Sara Silva, Stephen Dignum, and A. Sima Uyar, editors, *Proceedings of the 13th European Conference on Genetic Programming, EuroGP 2010*, volume 6021 of *LNCS*, pages 146–158, Istanbul, 7-9 April 2010. Springer.

Langdon, 2020a. W. B. Langdon. Genetic improvement of genetic programming. In Alexander (Sandy) Brownlee, Saemundur O. Haraldsson, Justyna Petke, and John R. Woodward, editors, *GI @ CEC 2020 Special Session*, page id24061, Internet, 19-24 July 2020. IEEE Computational Intelligence Society, IEEE Press.

Langdon, 2020b. W. B. Langdon. 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, October 2020.

Langdon, 2020c. W. B. Langdon. Multi-threaded memory efficient crossover in C++ for generational genetic programming. ArXiv, 22 September 2020.

Langdon, 2022a. W. B. Langdon. Genetic programming convergence. *Genetic Programming and Evolvable Machines*, 23(1):71–104, March 2022.

Langdon, 2022b. W. B. Langdon. Open to evolve embodied intelligence. In Fumiya Iida, Josie Hughes, Arsen Abdulali, and Ryman Hashem, editors, *Proceedings of 2022 International Conference on Embodied Intelligence, EI-2022*, volume 1292 of *IOP Conference Series: Materials Science and Engineering*, page 012021, Internet, Cambridge, 23-25 March 2022. IOP Publishing.

Langdon, 2022c. W. B. Langdon. A trillion genetic programming instructions per second. ArXiv:2205.03251, 6 May 2022.

Langdon, 2025. William B. Langdon. Improving a parallel C++ Intel AVX-512 SIMD linear genetic programming interpreter. ArXiv, 9 December 2025.

Lenski and others, 2015. Richard E. Lenski et al. Sustained fitness gains and variability in fitness trajectories in the long-term evolution experiment with Escherichia coli. *Proceedings of the Royal Society B*, 282(1821), 22 December 2015.

Maxwell III, 1994. Sidney R. Maxwell III. Experiments with a coroutine execution model for genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 413–417a, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.

Nordin *et al.*, 1995. Peter Nordin, Frank Francone, and Wolfgang Banzhaf. Explicitly defined introns and destructive crossover in genetic programming. In Justinian P. Rosca, editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 6–22, Tahoe City, California, USA, 9 July 1995.

Nordin *et al.*, 1999. Peter Nordin, Wolfgang Banzhaf, and Frank D. Francone. Efficient evolution of machine code for CISC architectures using instruction blocks and homologous crossover. In Lee Spector, William B. Langdon, Una-May O'Reilly, and Peter J. Angeline, editors, *Advances in Genetic Programming 3*, chapter 12, pages 275–299. MIT Press, Cambridge, MA, USA, June 1999.

Nordin, 1994. Peter Nordin. A compiling genetic programming system that directly manipulates the machine code. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 14, pages 311–331. MIT Press, 1994.

Oakley, 1994. Howard Oakley. Two scientific applications of genetic programming: Stack filters and non-linear equation fitting to chaotic data. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 17, pages 369–389. MIT Press, 1994.

Petke *et al.*, 2021. Justyna Petke, David Clark, and William B. Langdon. Software robustness: A survey, a theory, and some prospects. In Paris Avgeriou and Dongmei Zhang, editors, *ESEC/FSE 2021, Ideas, Visions and Reflections*, pages 1475–1478, Athens, Greece, 23-28 August 2021. ACM.

Poli and McPhee, 2013. Riccardo Poli and Nicholas Freitag McPhee. Parsimony pressure made easy: Solving the problem of bloat in GP. In Yossi Borenstein and Alberto Moraglio, editors, *Theory and Principled Methods for the Design of Metaheuristics*, Natural Computing Series, pages 181–204. Springer, 2013.

Poli *et al.*, 2008. Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008. (With contributions by J. R. Koza).

Syswerda, 1989. Gilbert Syswerda. Uniform crossover in genetic algorithms. In J. David Schaffer, editor, *Proceedings of the third international conference on Genetic Algorithms*, pages 2–9, George Mason University, 4-7 June 1989. Morgan Kaufmann.

Syswerda, 1990. Gilbert Syswerda. A study of reproduction in generational and steady state genetic algorithms. In Gregory J. E. Rawlings, editor, *Foundations of genetic algorithms*, pages 94–101. Morgan Kaufmann, Indiana University, 15-18 July 1990. Published 1991.

Tackett, 1994. Walter Alden Tackett. Greedy recombination and genetic search on the space of computer programs. In L. Darrell Whitley and Michael D. Vose, editors, *Foundations of Genetic Algorithms 3*, pages 271–297, Estes Park, Colorado, USA, 31 July–2 August 1994. Morgan Kaufmann. Published 1995.

Teller, 1994. Astro Teller. Genetic programming, indexed memory, the halting problem, and other curiosities. In *Proceedings of the 7th annual Florida Artificial Intelligence Research Symposium*, pages 270–274, Pensacola, Florida, USA, May 1994. IEEE Press.

Wang *et al.*, 2023. Richard J. Wang, Samer I. Al-Saffar, Jeffrey Rogers, and Matthew W. Hahn. Human generation times across the past 250,000 years. *Science Advances*, 9(1):eabm7047, 6 Jan 2023.

Woodward and Bai, 2009. John R. Woodward and Ruibin Bai. Canonical representation genetic programming. In Lihong Xu, Erik D. Goodman, Guoliang Chen, Darrell Whitley, and Yong-sheng Ding, editors, *GEC '09: Proceedings of the first ACM/SIGEVO Summit on Genetic and Evolutionary Computation*, pages 585–592, Shanghai, China, June 12-14 2009. ACM.