Convergence of Program Fitness Landscapes

W.B. Langdon

Computer Science, University College, London, Gower Street, London, UK
W.Langdon@cs.uc1.ac.uk
http://www.cs.ucl.ac.uk/staff/W.Langdon

Abstract. Point mutation has no effect on almost all linear programs. In two genetic programming (GP) computers (cyclic and bit flip) we calculate the fitness evaluations needed using steepest ascent and first ascent hill climbers and evolutionary search. We describe how the average fitness landscape scales with program length and give general bounds.

1 Introduction

Fitness landscapes are an attractive metaphor. Easy problems are supposed to have "smooth" landscapes, while hard problems are supposed to be caused by "rugged" landscapes. Much analysis of landscapes is empirical, but these have not led to general results in GP. [Kinnear, Jr., 1994] found little relationship between GP difficulty and ruggedness on a number of GP benchmarks, while [Nikolaev and Slavov, 1998,Clergue *et al.*, 2002] give counter examples. However [Daida *et al.*, 2001] agrees the landscape metaphor may be deceptive for GP. We conclude the empirical picture is not clear. As an alternative to amassing yet more data, we have analysed the general properties of GP, giving general results.

In [Langdon and Poli, 2002] we showed for both linear and tree genetic programming (GP) that in general the space of programs which GP searches converges as the programs get bigger. That is, beyond a threshold further increase in size makes little difference. Since then, for simplicity, we have concentrated on linear GP [Banzhaf *et al.*, 1998]. In linear GP, a program consists of a linear sequence (i.e. no loops) of instructions which manipulate memory registers. In [Langdon, 2002a] we defined five computer models (any, average, cyclic, bit flip and Boolean) and provided quantitative bounds on how long programs have to be so that the distribution of their outputs is near its limit. [Langdon, 2002b] deals quantitatively not only with program's outputs but also with the relationship between a program's outputs given different inputs, i.e. the function it implements. In [Langdon, 2003] we show that reversible programs tend to rather different limits. So far we have gathered formal results about the distribution of fitness of programs. This earlier analysis tells us about the search space but not how search operators connect it into a fitness landscape. Using the same computer models we use simple mutation in conjunction with two simple hill climbing strategies and population based search to yield results on the convergence of fitness landscapes and expected solution times.

E. Cantú-Paz et al. (Eds.): GECCO 2003, LNCS 2724, pp. 1702–1714, 2003.

In Sects. 3 and 4 we consider in detail the fitness landscape of two simple computers and calculate how long various search techniques will take to find programs of specified fitness. Section 5 shows we can put bounds on the fitness landscape for any computer, while Sect. 6 considers average behaviour across all computers. Some experimental measurements for the fifth model (Boolean linear GP) are given in Fig. 3.

2 Size of Neighbourhood – Point Mutation

Point mutation uniformly at random selects one instruction in a program and changes it. (So the mutant is always genetically different from its parent). Assume there are I different instructions, so a point mutation at a chosen point will convert the program to one of $I-1$ other programs. If the program contains l instructions, there are $l(I-1)$ other programs that can be created from it by a single point mutation. I.e., under point mutation each point in the fitness landscape has $l(I-1)$ distinct neighbours. Note the point mutation neighbourhood size increases directly in proportion to program size.

However since point mutation does not change program size, only a tiny fraction of all the programs can be reached by point mutation. Even as a fraction of programs of the same size, the neighbourhood is only $l(I-1)/I^l \approx l\ I^{1-l}$ of the total.

3 Cyclic Computer – Point Mutation Neighbourhood

The cyclic computer has only three instructions ($I = 3$) add one to memory, subtract one from memory, do nothing. Therefore the point neighbourhood size under point mutation is $l(I-1) = 2l$. The cyclic computer is obviously an unrealistic model of real computation, nonetheless by studying it we learn about real computers.

Suppose our program contains n_+ increments, n_0 no ops and n_- decrements. The complete point mutation neighbourhood contains n_- programs whose answers are two more (allowing wrap around), n_0+n_- whose answers are one more, n_++n_0 whose answers are one less and n_+ whose answers are two less. Note there are no neutral changes. The proportions of these four changes depends only upon the relative numbers of the three types of instructions in the original program and not directly on the length of the program. If we look at changes in program output, the point mutation landscape converges immediately to 1/6 -2, 1/3 -1, 1/3 $+1$ and 1/6 $+2$. Whereas if we consider the actual outputs exponentially large programs are needed for convergence [Langdon, 2002a].

If we start from a very unusual part of the search space ($n_+ \neq n_0 \neq n_-$) then this will distort the point mutation fitness landscape by changing the fractions of the four changes. If high fitness is associated with output values very different from those input, high fitness programs will have n_+ very different from n_-. This means point mutation of high fitness programs will on average produce offspring with lower fitness.

3.1 Hill Climbing with Point Mutation – Cyclic Computer

The simple instruction set means the output of any program is $(x + p) \bmod 2^m$. (Where x is its input, m is the number of bits in the output register and $p = n_+ - n_-$. So p is a constant for the program.) Note given the output for an input, the program's output for any other input can be inferred. Indeed we need only define one fitness case (e.g. input is zero). Define $F = |\text{output} - \text{target}|$, so the goal is to minimise F. We can calculate how long it will take two hill climbing strategies to find a solution.

In both cases, we start from a single randomly chosen program. For simplicity assume $n_+ = n_-$. (This is equivalent to assuming random fluctuations are small compared to the target. I.e. $\sqrt{l/3} \ll$ target. Also assume target $< 2^{m-1}$, so the fastest route is to increase p.) $p = 0$ initialy and the task is to find a sequence of point mutations which will increase p to target. If we use steepest ascent, we need only replace target/2 decrement instructions by increment instructions. This will take $2l \times \text{target}/2 = l \times$ target, fitness evaluations.

With first ascent, the fraction of mutations incrementing n_+ is $n_0/2l + n_-/2l$, while the fraction leading to a fall in n_- is n_-/l. Define $x = n_+/l$ and $y = n_-/l$ and treat x and y as continuous variables of the expected case. So

$$\frac{dx}{dt} = \tfrac{1}{l}\left(\tfrac{n_0}{2l} + \tfrac{n_-}{2l}\right) \Rightarrow x = 1 - \tfrac{2}{3}e^{-t/2l} \text{ and } \frac{dy}{dt} = -\tfrac{1}{l}\tfrac{n_-}{l} \Rightarrow y = \tfrac{1}{3}e^{-t/l}$$

The number of steps expected to be needed to first find a solution is given by the value of t for which target $= l(x - y)$. So target $= l(1 - \frac{2}{3}e^{-t/2l} - \frac{1}{3}e^{-t/l})$. Rearranging gives

$$t = -2l \log(2\sqrt{1 - 3 \text{ target}/4l} - 1) \text{ If target} \ll l \quad t \approx 3/2 \text{ target} \qquad (1)$$

That is, fitness initialy rises linearly but the rate of increase slows as the maximum fitness, given by the length of the program l, is approached.

If target $= l$ the problem becomes very similar to the OneMax problem. The approximation of treating x and y as continuous variables needs to be treated with care. Set target $= l - \epsilon$ in (1) and assuming $l \gg \epsilon$ and then let $\epsilon = 1$ gives t being in the region of $2l \log(2l/3)$. (Cf. $O(l \log l)$ for OneMax [Muhlenbein and Schlierkamp-Voosen, 1993].) In contrast steepest ascent requires l^2 fitness evaluations.

3.2 Population Approaches – Cyclic Computer

In the following analysis for simplicity we allow an offspring produced by point mutation into the population only if it is fitter than its parent. In which case, it replaces its parent.

Parallel Steepest Ascent. With steepest ascent each step explores the complete neighbourhood ($2l$ fitness evaluations) but, unless the problem has been solved, steepest ascent is guaranteed to find a better child, which will be accepted

by the population. Therefore the solution found will be the direct descendent of the fittest program (i.e. smallest F) in the initial population (fitness F_0). Since fitness is given by difference between a programs output and the required output and steepest ascent reduces the difference by 2 each generation, a solution will be found in generation $F_0/2$.

The number of n_+ in the initial (random) programs is given by a binomial distribution $C_{n_+}^{l-n_+}(\frac{1}{3})^{n_+}(\frac{2}{3})^{(l-n_+)}$. Even for modest l, this can be approximated by a Gaussian distribution with mean $l/3$ and variance $\frac{1}{3}\frac{2}{3}l = \frac{2}{9}l$. The initial distribution of n_0 and n_- are the same as that of n_+. Provided none of them is near zero, we can treat the distribution of any two as being independent, so the distribution of $p = n_+ - n_-$ can also be approximated by a Gaussian (whose mean is the difference in the means (0) and variance is sum of the variances $\frac{4}{9}l$, i.e. standard deviation $\sqrt{4/9l} = \frac{2}{3}\sqrt{l}$).

The likely fitness of the fittest program in the initial population is given by the population size, M_l, $F_{0l} = \text{target} - \frac{2}{3}\sqrt{l}\,\Phi^{-1}(1-1/M_l)$. ($\Phi^{-1}$ is the inverse of the integral of the Gaussian distribution, it gives the number of standard deviations for a particular probability.) Note that both larger initial programs and a larger population can be expected to give a fitter initial best program. The expected number of generations to find a solution is $\text{target}/2 - \max_l 1/3\sqrt{l}\,\Phi^{-1}(1-1/M_l)$. If the initial population is large, so that there are a large number of programs of each length, we can be confident that the fittest program in the initial population is also one of the longest. Further that $\Phi^{-1}(1-1/M_l) \approx 3$. Therefore the expected number of generations to find the first solution will be about $\text{target}/2 - \sqrt{l_{\max}}$.

The number of fitness evaluations depends upon the spread of initial fitness values and the selection technique used. We assume the selection pressure is strong enough to ensure at least one copy of the fittest program is copied to the mating pool. With tournament selection and tournaments of size T there will be on average T copies of the best. This leads to rapid convergence of the population (in $\approx \log_T M$ generations[1]). If $\log_T M \ll \text{target}/2 - \sqrt{l_{\max}}$ then the number of fitness evaluations expected to be required to find a solution will be about $Ml_{\max}(\text{target} - 2\sqrt{l_{\max}})$. Rather fewer fitness evaluations will be needed if the population still retains shorter programs.

At the other extreme is to have no selection pressure and instead to give each member of the current population exactly one child. With steepest ascent, each offspring will be exactly 2 fitter than its parent and so all children will be inserted into the next generation. Therefore the number of fitness evaluations expected to be required to find a solution will be about $M\bar{l}(\text{target} - 2\sqrt{l_{\max}})$. Where $\bar{l}$ is the mean length of programs, in the initial and hence every generation.

Notice how the temporal granularity of having fixed non-overlapping generations gives rise to a simple population dynamics. Suppose instead of forcing each steepest ascent in the population to synchronise by waiting for every new child, we allow each child to be compared with its parent immediately. This gives a speed advantage to shorter programs. Using tournament selection we now get a race. A few of the longer programs initialy have an advantage and we can

[1] [Goldberg and Deb, 1991, p74 and p80] includes an additional $\log_T \log M$ term.

expect the average length of programs to start to increase. What happens next depends in a complicated way on the distribution of program lengths and the selection pressure. If the selection pressure is very high and the programs are of similar lengths then we expect shorter programs to be removed from the population before they can catch up with the fittest (longest) program and the average program size will continue to increase. However if the shorter programs are very much shorter and the selection pressure is not so great, one of them can increase its fitness much faster than the longest and so then will be selected for, causing the average program length to decrease.

Since the second selection scheme (where every program gets exactly one child, and hence is replaced by it) is simpler, we can analyse it in more detail. Assume all programs are at least long enough to be able to solve the problem. After tM fitness evaluations the fitness of the best program of length l will be about target $- 2\sqrt{l} - t/l$. (Remember its initial fitness $\approx$ target $- 2\sqrt{l}$ and it improves by $+2$ every $2lM$ fitness evaluations.) Initialy the best program in the population will also be one of the longest but the shortest will catch it up. We can calculate the expected number of fitness evaluations t needed by setting the expected best fitness of the longest and shortest programs to be equal. Define $r = l_{\max}/l_{\min}$ then

$$2\sqrt{l_{\min}} + t/l_{\min} = 2\sqrt{l_{\max}} + t/l_{\max} \Rightarrow t = 2\frac{1-r^{-0.5}}{r-1}\ l_{\max}^{3/2}$$

If target is small then it will be found first by one of the longest programs, otherwise by one of the shortest. Substituting t we get the critical target value $\text{target}_{\text{crit}} = 2\left(\frac{r-r^{-0.5}}{r-1}\right)\sqrt{l_{\max}}$. If target is less than the critical value the number of fitness evaluations expected to solve the problem is about $Ml_{\max}(\text{target} - 2\sqrt{l_{\max}})$ and $Ml_{\min}(\text{target} - 2\sqrt{l_{\min}})$ otherwise.

Parallel First Ascent. With first ascent mutation in a population it is natural to consider a generational approach in which every offspring is produced by exactly one point mutation. The M new individuals then become candidates to be members of the new population. Note this finer level of granularity removes the speed advantage of shorter programs seen with steepest ascent (previous section). In fact longer programs now have a modest advantage since as fitness climbs the chance of making a successful point mutation falls more slowly than it does for the shorter programs, cf. derivation of Equation (1).

With tournament selection we would expect rapid convergence of program sizes towards that of the fittest individual in the initial population. (With a large population we expect this to be the longest length in the initial population.) I.e. the average program size will grow towards $l_{\max}$ in $\approx \log_T M$ generations.

If the selection pressure is high ($T \gg 2l/n_- \approx 6$) then we can be reasonably sure each generation at least one of the T children of the best individual in the population will have been formed by mutating a decrement instruction into an increment instruction, increasing its output by 2 compared to its parent. Thus the same number of generations, $\text{target}/2 - \sqrt{l_{\max}}$, will be needed to solve the

problem, as are needed by steepest ascent. The number of fitness evaluations will be $M(\text{target}/2 - \sqrt{l_{\max}})$. If the fittest is not quite so dominant, after each improvement is found their may be a gap generation where the next improvement is not found or a smaller improvement in fitness is found. The existing best will spread through the population, giving it about T^2 copies on average in the next generation, making it much more likely a +2 fitness improvement will be found. Hence $\text{target}/2 - \sqrt{l_{\max}}$ is a reasonable estimate even for more modest tournament sizes.

In the second selection scheme (in which every program gets exactly one child, which is the same length as it) there will be no change in average program size. The second selection scheme means each slot in the population is independent and so hill climbs in parallel but in isolation. Taking into account the expected best fitness in the initial generation ($\approx 2\sqrt{l_{\max}}$) in Equation (1), gives the average number of fitness evaluations required to solve the problem as $\approx -2Ml_{\max}\log(2\sqrt{1 - 3\ (\text{target} - 2\sqrt{l_{\max}})/4l_{\max}} - 1)$. With M searches in parallel we can expect one lucky one to find a solution before the others. If this were included, the number of fitness evaluations would be reduced by $\mathrm{O}(M\sqrt{t})$. Where target $\ll l_{\max}$ the number of fitness evaluations to reach target is about $3/2\ M(\text{target} - 2\sqrt{l_{\max}})$, cf. Approximation (1).

Notice that even though we have used a single genetic operator on the same fitness function, i.e. a single fitness landscape, we have seen many different behaviours. Small differences in the sequence of fitness evaluation, selection and replacement can lead to macroscopic changes. "One operator, One landscape" [Jones, 1995] is not sufficient to explain evolution.

4 Bit Flip Computer

Our second example is the bit flip computer. It contains N bits of memory and $N + 1$ instructions, one no op and N instructions which read their memory cell and invert it. All N bits can be used during a program's execution but only the m bits of the output register (which overlap the input register) are used for output when the program stops. Like the cyclic computer, it can only implement 2^m functions and in the long program limit each are equally likely. However they need only contain $\frac{1}{4}(N + 1)(\log(m) + 4)$ random instructions to be close to the uniform limit, rather than an exponentially large number for the cyclic computer [Langdon, 2002a].

If we follow any program by $\frac{1}{4}(N + 1)(\log(m) + 4)$ randomly chosen instructions its new fitness will effectively be uncorrelated with its original fitness. Adjacent pairs of instructions flipping the same bit can be stripped out of the random addition with no effect. Each remaining random i,j pair has the same effect as replacing an i instruction in the original program with a j instruction. (If the program did not contain any i instructions, two can be created by mutating a pair of two other instructions, which need not be adjacent, into i instructions.) I.e. no more than $\frac{1}{4}(N + 1)(\log(m) + 4)$ independent point mutations are needed to scramble any bit flip computer program's fitness.

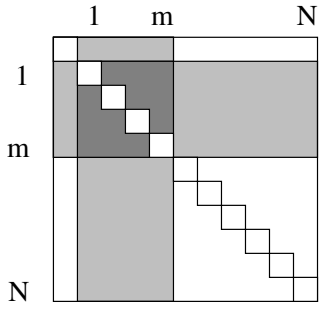

Fig. 1. Effect of point mutation on bit flip computer programs. Original instruction (rows) v. new instruction (columns). White – not a mutation or no effect, light grey – 1 bit flipped, dark – 2 output bits flipped.

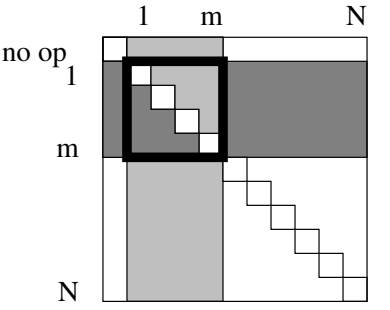

Fig. 2. Acceptance by first ascent hill climber. White – not a mutation or not accepted, light grey – accepted if no. new instructions is even, dark – accepted if no. old instructions was even. Inside bold square mutation will change fitness by two bits, elsewhere only one.

Unlike the cyclic computer, many point mutations have no effect. Of the $N(N+1)$ mutations from one instruction to another, $(N-m)(N+1-m)$ affect only the $N-m$ bits of memory not used by the output register and so have no effect on the value output by the program. $2m(N+1-m)$ invert one bit of the output register, while the remaining $m(m-1)$ invert two output bits (see Fig. 1).

For concreteness we define the fitness function to be given uniquely by the function implemented by each program. For this computer, this is equal to the value returned by the program when given input zero. Which in turn is given by counting the number of instructions flipping bits $1 \ldots m$. Call each of these C_i, and then the formula for fitness is fitness $= \sum_{i=1}^{m} (C_i \bmod 2) 2^{i-1}$. In the long program limit, each fitness is equally likely, so the mean fitness is $2^{m-1} - 0.5$ with variance $\sqrt{(2^{2m}-1)/12}$ (SD $\approx 0.5774\ 2^{m-1}$).

The fitness neighbourhood under point mutation of a program depends upon the relative number of each of the instructions from which is made, i.e. not just on its fitness. For example, a program of $2N$ no ops will have fitness 0 and can be mutated to fitness $0, 1, 2, 4, \ldots 2^m$. (Looking at the top row of Fig. 1 we see the probability of no fitness change is $(N-m)/N$ and that of each change is $1/N$.) While another program with N pairs of each bit flip operations will also have fitness 0. It too can be mutated to fitness 0 (cf. both lower white areas in Fig. 1, probability $((N-m)/N)^2$), and to $1, 2, 4, \ldots\ 2^m$. (Each 1 bit change has probability $(1+2(N-m))/N^2$, cf. light grey in Fig. 1). But also to fitnesses $3, 5, 6, 9, 10, \ldots\ (2^{m-1}+2^m)$. (Each 2 bit change has probability $2/N^2$, dark grey in Fig. 1). These two examples show, that the current fitness of a program is not sufficient to define either its neighbours or the probability of moving between particular points on the fitness landscape.

4.1 Bit Flip Computer – Steepest Ascent

We start with a randomly chosen program of length l ($l \geq m$). About half the C_i will be odd. Guided by the fitness function, steepest ascent hill climbing will take about $m/2$ complete steps to find a program with maximum fitness $2^m - 1$. Each step takes lN fitness evaluations. I.e. we expect to reach the optimum in $\frac{1}{2}lmN$ fitness evaluations. If we chose an initial program of exactly the minimum length ($l = m$) then the effort is minimised $\frac{1}{2}m^2N$.

4.2 Bit Flip Computer – First Ascent

Again we start with a randomly chosen program of length l. For simplicity we will assume exactly $m/2$ of C_i ($1 \leq i \leq m$) are odd. Otherwise the instructions are uniformly chosen. We make random point mutations one at a time. After each the offspring's fitness is calculated and the mutation is accepted if its fitness is greater than before. See Fig. 2.

If $l \gg m$ we can assume that even as the search proceeds and changes are made to the program, the proportion of each instruction remains nearly uniform. Therefore each of the $N(N+1)$ possible mutations (cf. Fig. 1) remain equally likely. However the chance of a mutation being accepted falls as more instructions become correctly paired. Unless both the instruction being replaced and the one replacing it affect the output register (i.e. both lie inside the inner square of Fig. 2) the chance of acceptance will remain proportional to the number bits unset in the fitness value. Inside the square, mutations affect two bits of the fitness. Accepted mutations may set them both or set the larger one but clear the smaller. For simplicity we ignore the second possibility, this means we will slightly over estimate the average number of mutations needed to reach a solution.

The probability of an accepted mutation which increases the number of $C_{1\leq i\leq m}$ which are odd by one is $\frac{\text{even}}{m}\frac{2m(N+1-m)}{N(N+1)}$. Where even is the number of $C_{1\leq i\leq m}$ which are even. While the chance of an accepted mutation increasing the number of C_i which are odd by two is about $\left(\frac{\text{even}}{m}\right)^2\frac{m(m-1)}{N(N+1)}$. Combining gives the average reduction in the number of mismatched I/O flips per mutation as $\geq 2\frac{\frac{\text{even}}{m}m(N+1-m)+\left(\frac{\text{even}}{m}\right)^2 m(m-1)}{N(N+1)}$. So the expected number of fitness evaluations required to find a solution is

$$\leq \frac{N(N+1)}{2}\sum_{\text{even}=m/2}^{1}\frac{1}{\frac{\text{even}}{m}m(N+1-m)+(\frac{\text{even}}{m})^2 m(m-1)}$$

$$= \frac{N(N+1)}{2}\sum_{\text{even}=m/2}^{1}\frac{1}{\frac{\text{even}}{m}m(N+1-m)} - \frac{1}{\frac{\text{even}}{m}m(N+1-m)+\frac{m^2(N+1-m)^2}{m(m-1)}}$$

$$\approx \frac{N(N+1)}{2(N+1-m)}\left(H\left(\frac{m}{2}\right)-H\left(\frac{m}{2}+\frac{m(N+1-m)}{(m-1)}\right)+H\left(1+\frac{m(N+1-m)}{(m-1)}\right)\right)$$

H is the Harmonic Number, $H(x) = \sum_{i=1}^{x} 1/i$. If $x \gg 1$, $H(x) \approx \log x + \gamma$, where γ is Euler's constant (≈ 0.57721566). So assuming $N \gg m \gg 1$, the

expected number of fitness evaluations required to find a solution is about $\frac{1}{2}N\log(0.8905362\ m)$. Note this means on average first ascent requires fewer fitness evaluations than than steepest ascent.

5 Any Irreversible Computer

By an "irreversible" computer we mean that there is a program which when run with two or more inputs yields the same answer. I.e. it is impossible to run the program backwards from its output to uniquely determine what input it was initialy given. Practical computers are irreversible[2].

Consider two or more identical copies of a computer. They run copies (or mutated copies) of the same program and their clocks are synchronised. They may have different initial conditions but we shall show after running a long randomly chosen program they will all become synchronised. Once two such computers are synchronised they will remain synchronised (unless one of them strikes a mutation).

We strengthen our definition of "irreversible" to require that for every pair of states there exists a computer program which when run on two computers, each starting in one of the states, will eventually cause both to be in the same state at the same time. Define a as being the length of the longest (over all pairs of states) such minimal program.

Suppose we chose a program at random and evaluate its fitness, i.e. run it on each of T fitness cases. Then we mutate it and evaluate the fitness of its offspring. In both cases the computer goes through a random sequence of states until the program reaches its last instruction and halts. For an arbitrary mutation, after a instructions past the mutation site, it is possible that both programs will arrive at the same state. (By definition, at least one of the I^a, sequences of a instructions, will do this.) In which case, they will remain synchronised and so output the same value. If for each fitness case, they always synchronise then they will have the same overall fitness.

If the mutation is followed by a instructions the chance the two programs will finish in the same state is at least I^{-a}. Suppose that just before the mutation site, over the T pairs of runs, the computers are in T' distinct states ($T' \le T$). If the mutation is followed by $T'a$ instructions, the chance the two programs will finish together in the same state when run in T pairs is at least $T'!I^{-T'a}$. If the mutation is followed by i random instructions the chance of being in different states for at least one input is no more than $(1-T'!I^{-T'a})^{\lfloor i/T'a \rfloor}$. For a program of length l the average chance of a mutation causing a fitness change is no more than

$$\le \frac{1}{l}\sum_{i=T'a}^{l-1}(1-T'!I^{-T'a})^{\lfloor i/T'a \rfloor} \ < \frac{T'aI^{T'a}}{l}\left(1-(1-T'!I^{-T'a})^{l/T'a}\right)$$

The average chance of a point mutation changing the fitness of a long average program falls at least in proportion to its length. For any irreversible computer,

[2] In contrast quantum computers are reversible.

and any fitness function, almost all (i.e. at least 90%) point mutations on programs longer than 10 TaI^{Ta} have no effect.

5.1 Average Fitness on Any Computer

We shall show on any irreversible computer almost all programs are useless. Again we run each of the T fitness cases on T copies of the same program running on identical computers. If the program is chosen at random and is at least a instructions long then there is a chance of at least $(T-1)I^{-a}$ that two of the computers will yield the same output. If we make it aT instructions long the chance all T computers are synchronised is at least $(T-1)!I^{-Ta}$. So the chance of not returning a constant is less than $1-(T-1)!I^{-Ta}$. For programs of length $l \geq Ta$ the chance of not returning a constant is $\leq (1-(T-1)!I^{-Ta})^{\lfloor l/Ta \rfloor}$. Setting this to 10%, taking logs and rearranging gives a lower bound, meaning almost all programs longer than $2.3\ TaI^{Ta}/(T-1)!$ when run all T fitness cases the program will yield the same output. If the fitness testing is exhaustive, (all 2^n tests are run) then on any irreversible computer almost all programs longer than $2.3\ 2^n aI^{2^n a}/(2^n-1)! \approx 0.85\ a(2.718\ I^a/2^n)^{2^n}$ have zero fitness.

6 Average Computer

In the previous sections we have been treating the computer as a machine whose state is given by its memory and movement between those states is controlled by instructions in its instruction set. In two cases we have considered a specific class of computers and this prescribed their instruction set, while in the other we have set very loose restrictions on the computer to derive general limits for all computers. We define an average computer as one that is representative of all the computers with linear programs (no loops), a fixed memory N bits and fixed number of instructions I. The trick is to realise that a random change is by far the most likely of all the possible ways an instruction could change the computer's memory. I.e. the average computer contains I instructions each of which makes fixed but random changes to the value held in its memory.

On the average computer point mutation is very disruptive. Changing a single instruction means that just after the mutation the state of a computer running the parent program and that of one running the mutation are totally uncorrelated.

6.1 Fitness of the Average Computer Program

Here we take a very high level definition of fitness. We say if a program is run on T test cases and yields T' different answers then its fitness is T'. As before we set up T copies of the computer and program, all running in lock step. Assume $2^N \gg 2^m \gg T \gg 1$. If $l \leq 2^N/(T-1)$ then on average all of the T computers are in different states and so the expected fitness $\overline{f}$ is T. If l is between $2^N/(T-1)$ and $2^N/(T-1)+2^N/(T-2)$ then on average two of

them are in the same state so $\overline{f}$ is $T-1$. In general, if $l \leq 2^N \sum_{i=T'-1}^{i=T-1} 1/i$ then $\overline{f} \geq T'$. Approximating this sum (the Harmonic number) with the logarithm and rearranging gives the expected fitness $\overline{f} \approx 1 + (T-1)e^{-l/2^N}$, indicating on average fitness falls exponentially with program length. (This is consistent with the expected length at which programs become independent of their inputs $(\log(T-1)+\gamma)2^N$ [Langdon, 2002b, 4.2].)

6.2 Average Computer – Point Mutation

Suppose we set up T pairs of identical computers running in lock step. Initialy all run the same program. Now suppose we make the same point mutation to one program of each pair. As the computer pairs run, initialy they will be in the same state but overall we expect the total number of different states to start at T and then fall exponentially (as described in Sect. 6.1). Suppose at the point of mutation, the total number of different states of the $2T$ computers is T'. Just after the point where the programs run into the mutation, it is very likely that each pair will separate so doubling the total number of different states to $2T'$. However if the programs run on after the mutation site, the number of different states falls exponentially. Averaging across all l mutation sites, we can approximate the expected number of different states following a point mutation (at instruction x) when a program of length l terminates as

$$\frac{1}{l}\int_0^l 1+\left(1+2(T-1)e^{-x/2^N}\right)e^{-(l-x)/2^N}dx = 1+2(T-1)e^{-l/2^N}+\frac{2^N}{l}\left(1-e^{-l/2^N}\right)$$
$$\approx 1+\frac{2^N}{l} \qquad \text{if } l \gg 2^N$$

In small programs ($l \ll 2^N$) this is approximately $2Te^{-l/2^N}$. I.e. on average a single point mutation to a short program is very disruptive but for very long programs its impact on the program's fitness falls as $\mathrm{O}(l^{-1})$.

6.3 Non-reversible Computer – Crossover

Two point crossover between two average programs essentially means inserting a randomly selected code fragment into one program and the corresponding fragment into the other parent. The situation is slightly more complex than with point mutation. For example the length of the code changed aught to be considered. However on the average machine, inserting a random code fragment will have much the same effect as a single random instruction change.

With uniform crossover even with very long programs there will be coding changes near each end of the program. The effect becomes very similar to replacing the whole of the parent program with another randomly selected one. Therefore we would expect no correlation between the outputs of the parent and child program. In the case of the average computer, the expected number of different outputs they produce with T input test cases will be the same. (Since

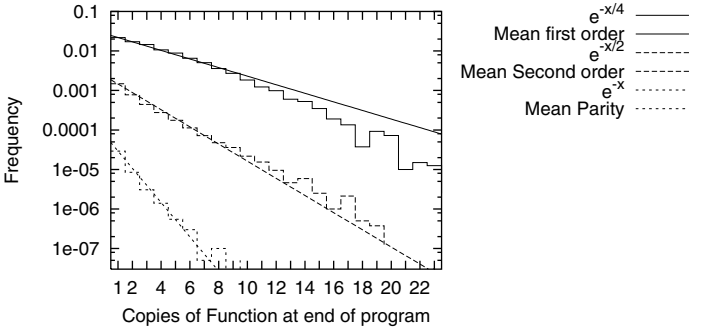

Fig. 3. Proportion of programs (1000 instructions, 128 bits) containing multiple copies of each function. The functions fall into four classes, constants (not shown), copies of inputs, second order and parity.

it is given by their length, which, depending upon the crossover operator, will be the same.) I.e. with this fitness function, we expect their fitness to be about the same.

7 Conclusions

We have proved general results. Fitness landscapes do converge. Most programs are useless, and mutating them is unlikely to improve them. How do we reconcile this with GP? How do we progress? Here the results have been for linear non reversible programs, which loose information. However other representations, which do not (e.g. tree GP, reversible computing and linear GP with inputs write protected) also converge. Nor should we hope to concentrate of programs smaller than the convergence threshold, since this can still be a very large number of programs.

How does Nature do it? When we look at evolved organisms we see tremendous reuse of partial solutions, hierarchies and modularity. Most programs are amorphous. The fitness landscape as a whole is dominated by these useless programs. If the fitness landscape metaphor is to help, like any map, it needs to concentrate on routes to where we want to be. May be future analysis will shed light on the structure of good programs and the route map created by crossover.

Acknowledgements. I would like to thank Ben Dias.

References

[Banzhaf *et al.*, 1998] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming – An Introduction.* Morgan Kaufmann.

[Clergue *et al.*, 2002] Fitness distance correlation and problem difficulty for genetic programming. In *GECCO 2002*, pp 724–732, New York, 9-13 July 2002.

[Daida *et al.*, 2001] Jason M. Daida, *et al.*. What makes a problem GP-hard? *Genetic Programming and Evolvable Machines*, 2(2):165–191, June 2001.

[Goldberg and Deb, 1991] A comparative analysis of selection schemes used in genetic algorithms. In G. J. E. Rawlins, editor, *FOGA*, pp 69–93. Morgan Kaufmann.

[Jones, 1995] Terry Jones. One operator, one landscape. Technical Report SFI TR 95-02-025, Santa Fe Institute, January 1995.

[Kinnear, Jr., 1994] Kenneth E. Kinnear, Jr. Fitness landscapes and difficulty in genetic programming. In *WCCI*, pp 142–147, Orlando, 27-29 June 1994. IEEE Press.

[Langdon and Poli, 2002] W. B. Langdon and Riccardo Poli. *Foundations of Genetic Programming*. Springer-Verlag, 2002.

[Langdon, 2002a] Convergence rates for the distribution of program outputs. In *GECCO 2002*, pp 812–819, New York, 9-13 July 2002. Morgan Kaufmann.

[Langdon, 2002b] W. B. Langdon. How many good programs are there? How long are they? In Jonathan Rowe, *et al.* editors, *FOGA VII*. Morgan Kaufmann.

[Langdon, 2003] W. B. Langdon. The distribution of reversible functions is Normal. In Rick Riolo, editor, *GP Theory and Practise*. 2003. Forthcoming.

[Muhlenbein and Schlierkamp-Voosen, 1993] Predictive models for the breeder genetic algorithm. *Evolutionary Computation*, 1(1):25–49, 1993.

[Nikolaev and Slavov, 1998] Concepts of inductive genetic programming. In W. Banzhaf, *et al.* editors, *EuroGP*, *LNCS* 1391, pp 49–60. Springer-Verlag.