Repeated Sequences in Linear GP Genomes

W. B. Langdon and W. Banzhaf

Computer Science, Memorial University of Newfoundland, Canada

Abstract. Biological chromosomes are replete with repetitive sequences, microsatellites, SSR tracts, ALU, etc. in their DNA base sequences. We discover hierarchical repeating sequences (building blocks?) are evolved by genetic programming in linear time series prediction programs.

"DNA whose sequence is not maintained by selection will develop periodicities as a result of random crossover" George P Smith, Science, 1976.

1 Introduction

It has been long noticed that there are emergent phenomena in genetic programming (GP) runs unintended by the human designer of the algorithm. Early on it was observed that code which does not change the output of the program (i.e. non-effective code) appears in many GP runs [26,29,2]. It was also noted that bloat affects many GP systems. Reasons for bloat and non-effective code have been examined in years past [21,4,5] and remedies have been developed more or less effective under particular circumstances. (e.g. [23,13,18,14]).

Here we would like to argue that non-effective code and bloat are only the tip of an iceberg and that there is more to be discovered about "emergent phenomena" in GP runs. Particularly, we would like to study repetition of patterns in GP-evolved programs. These are instructions, or more interestingly, groups of instructions, that occur several times in a program. In fact long sequences of instructions which are repeated can sometimes be decomposed into shorter repeated sequences. Although this is interesting in itself, it parallels what has been found in natural genomes. Biologists have long noticed the curious existence of repeated sequences in genomic DNA.

Perhaps the reasons for emergence of repeated sequences is similar in biological and artificial evolutionary systems? What could we learn from biological explanations, and can we transfer understanding from Evolutionary Algorithms back into Biology? What instruments are available for observing and examining repetitive sequences? Are there new representations of GP that might be more conducive to evolution once the reason for emergence of repeated sequences has been understood? Are we on the way to discover that evolution reuses code in a very interesting, yet hardly intelligible way? Are building blocks involved in the formation of repeated sequences? These and more questions are raised by our observations.

We first discuss the biological background to repeated sequences. Section 3 describes the linear GP system used for our experiments and the time series prediction task it was applied to. Section 4 presents results of our experiments while Section 5 concludes and outlines future research.

2 Biological Background

Biologists have discovered that there is a vast amount of repetition in the DNA of microbes, plants and animals [10]. Given that less than 3% of a human genome consists of protein-coding genes and about 50% of it consists of repetitive sequences, many of viral origin [27,25], initially Biologists concentrated upon understanding of the information content of expressed part of genomes. With whole genome analysis becoming more prevalent in recent years, the ubiquity of repetitive DNA is a lively subject of research [22,31,1]. There are various forms of repeated DNA, and the multitude speaks to the fact of a complex phenomenon: There are satellites, mini-satellites and micro-satellites, repeats of different sizes located next to each other along the genome, there are ALU repeats and interspersed repetitive sequences, both in coding, non-coding and intercistronic areas. Repeats are well distributed over genomes and species, and constitute a considerable fraction of all DNA in organisms.

The search for causes began some time ago. Smith, in 1976, did numerical experiments in order to explain evolution of repeated DNA sequences [28]. His conclusion was that homologous crossover is a major factor in the emergence of repeated sequences. In more recent work crossover and DNA duplication have been identified as important factors. Driven by the inaccuracy of the DNA replication machinery, repeated sequences are both a consequence of misalignments and a cause for crossover [11]. Hsieh and Lee considered a model of bacterial genome growth working with a mechanism called "random segmental self-copying" [15]. This model was able to explain, at a statistical level, the distribution of patterns found in bacterial genomes. They concluded that growth processes of genomes must have taken place, as the statistical traces of these are still visible in the distribution of DNA patterns.

In recent years, quantitative analysis tools have become available in molecular biology that allow a closer look at these phenomena [7,8,30]. This will provide the opportunity to observe even more closely how many different repetitive patterns emerged during evolution of a particular genome. At the same time, genetic applications of repetitive sequences are beginning to appear [16] which promise to facilitate research in experimental settings.

3 The Linear Genetic Programming System

A standard linear genetic programming system (GPengine) was used for our experiments. Sections 3.1 to 3.5 describe its operation. Section 3.6 describes the benchmark used (predicting the chaotic Mackey-Glass time series). Cf. Table 1.

3.1 Tournament Selection and Steady State Population

GPengine uses a steady state population and tournament selection. Four distinct individuals are chosen at random[1] from the population. The fitness of the first two are compared, giving a winner and a looser. In the event they have identical root mean squared (RMS) error, the tie is broken arbitrarily. The second pair are compared in the same way to give a second winner and looser.

[1] GPengine uses the C rand function.

Table 1. GP Mackey-Glass Parameters

Objective:	Evolve a prediction for a chaotic time series
Function set:	$+ - \times \div$ (operating on unsigned bytes) If 2^{nd}arg $= 0, \div = 0$.
Terminal set:	8 read-write registers, constants 0..127. Registers are initialised with historical values of time series. R0 128 time steps ago, R1 64, R2 32, R3 16, R4 8, R5 4, R6 2 and finally R7 with the previous value. Time points before the start of the series are set to zero.
Fitness:	Root mean error between GP prediction (final value in R0) and actual (averaged over 1201 time points).
Selection:	Steady state, tournament 2 by 2
Initial pop:	Random program's length uniform chosen from 1..14
Parameters:	Population 500, Max Program Size 500, 90% crossover, 40% mutation
Termination:	125500 individuals evaluated

Output R0..R7	Arg 1 R0..R7	Opcode $+ - * /$	Arg 2 0...127 or R0..R7

Fig. 1. Format of a GPengine instruction.

The offspring produced from the two winners (by crossover, mutation or copying (cloning)) replace the two losers. Note each tournament always produces exactly two children and the same method is used to produce both children.

Using this form of tournament selection in a steady state population means the best in the population cannot get worse. However the best individual is not immortal. If more than one individual has the smallest RMS error, the best individual may by chance be deleted (and replaced by the offspring of one of the other individuals which also had the smallest error).

3.2 GP Representation and Evaluation

Each individual consists of a linear sequence of instructions. Each instruction takes two inputs, performs its calculation and writes the output to a register. The first input is always a register. The second can either be a constant (0..127) or a register. Figure 1 describes a single instruction. We use eight 8-bit read-write registers. Before the individual is executed, all the registers are initialised with data for the current fitness case. The sequence of instructions is obeyed from the start of the individual to its end. The final value in register R0 is the GP's output, i.e. its prediction.

3.3 Crossover

90% of tournaments are followed immediately by crossover of the two winners, yielding two children which overwrite the two losers. In the other 10% of cases, the losers are overwritten by copies of the winners. Two-point crossover is used (see Figure 2) however GPengine appends to the end of first parent if the code to be copied from the second does not overlap the first. For this to happen the
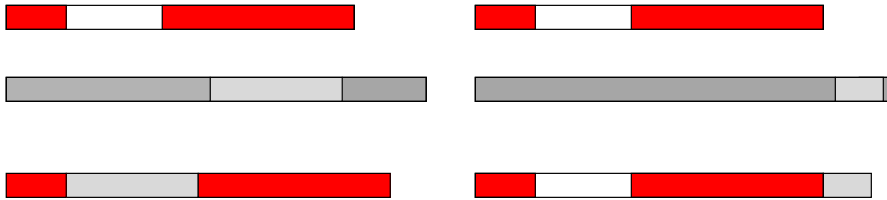
Fig. 2. Two instructions are randomly chosen in each parent (top two genomes) as cut points. If the code to be inserted from the second parent at least partially overlaps the first, it is inserted in the normal way to give the child (lower chromosome). With headless chicken crossover, the inserted code is randomly generated.

Fig. 3. XOA crossover. If there is no length overlap between code selected in second parent and the first parent (top), the selected code fragment is appended to the whole of the code from the first parent. (I.e. the middle portion of the first parent in not removed.)

second parent must be longer than the first (see Figure 3). In a second set of experiments this append variation was disabled (and insertion was used in all cases). If the chosen crossover points would mean the potential offspring would exceed the maximum size (500 instructions) then the crossover is aborted and the looser is not overwritten. Note the length checks are made independently, so the other crossover may proceed. Even if the looser is not replaced by crossover, it may still be changed by mutation.

Note that in GP we take it for granted that the parent programs are aligned at their starts. This provides a huge degree of both syntactic and semantic homology for free. This is similar to Nature, where chromosomes are crossed on a like-for-like basis. But at the detailed level where natural crossover occurs, Nature has to work to find matching DNA sequences to establish crossover points.

3.4 Variable length Mutation – Headless Chicken Crossover

The initial programs are quite short. In order to study if crossover was uniquely responsible for repeating sequences we used a mutation operator which could change program lengths. We introduce headless chicken crossover (HCX) [3] to linear GP. Although described as a crossover operation, only the length of the second parent has any influence on the child. Initially HCX works in the same way as two point crossover (see Figure 2) except that instead of inserting a code fragment taken from the second parent a randomly generated sequence of code *of the same length* is inserted.

GPengine does not write protect its inputs, this means a long sequence of random instructions will eventually overwrite all the registers. Since the instructions are not reversible, each overwrite destroys information. If the random sequence is long enough it is virtually guaranteed to destroy all information in the registers.

Once that happens a program's initial conditions cannot affect its subsequent behaviour. Such programs are useless at predicting and so have large RMS errors. Assuming each overwrite is 100% destructive, a random sequence of about $8(\log 8 + \gamma) \approx 21.3$ instructions will render the offspring useless [19]. The expected size of the crossover fragment is $\frac{1}{l} \sum_{i=0}^{l-1} \frac{1}{2}(l-i) = (l+1)/4$, where l is the number of instructions in the second parent. Hence we anticipate runs with only headless chicken crossover will not bloat much above 84 instructions.

When the second program is long enough, headless chicken crossover becomes like a supersonic jet nozzle. Flow downstream of the nozzle is independent of that upstream. Similarly program outputs (which are downstream of the random code) are independent of inputs. I.e. they are disconnected from upstream perturbations.

3.5 GP Mutation

After two children have been produced by crossover or by simply copying their parents (cloning), there is a 40% chance that they will be mutated. Mutation consists of choosing uniformly at random exactly one instruction in the individual and changing it. Each of the four fields in the chosen instruction (cf. Figure 1) is equally likely to be changed. Apart from ensuring the new instruction is different, the mechanism is the same as that used to create the initial population. Note this means the second argument is approximately equally likely to be a constant (0..127) or a register (R0..R7). The other three fields are chosen uniformly from their legal values.

3.6 Mackey-Glass Benchmark

Since the goal was to study the long term behaviour of an evolving population of programs we need a moderately difficult task. The population should continually improve and neither get stuck because the problem is too hard nor quickly find the optimal solution. We chose the problem of time series prediction as this is both hard and interesting. Indeed it has applications in scientific, medical and financial modeling [24]. We used the IEEE benchmark Mackey-Glass chaotic time series (`http://neural.cs.nthu.edu.tw/jang/benchmark/`, $\tau = 17$, 1201 data points, sampled every 0.1)). 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). See Figure 4.

The task for the GP is, given historical data, to predict the next value[2]. The GP is given eight values from earlier in the series. Arguably the most useful is that from the previous time step (which is loaded into R7) but values 2 time steps ago, 4, 8, 16, 32, 64 and 128 time periods back are also available. As with the benchmark, values before the start of the sequence are set to zero. Note that the GP system only has eight byte registers, and if it needs scratch registers, it may have to sacrifice one or more inputs to store intermediate results.

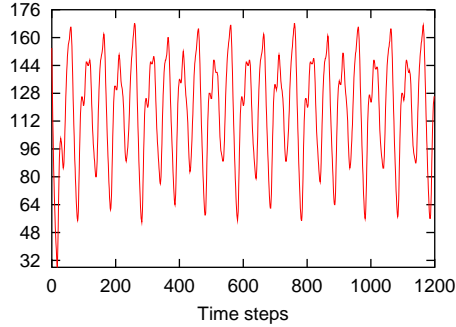[2] Since the series is chaotic this cannot be done exactly.

Fig. 4. Discrete Mackey Glass chaotic time series

Table 2. Best Mackey Glass prediction error at end of runs: using two point crossover with append (XOA), without append (2XO) and headless chicken crossover crossover (HCX). With and without fitness selection.

	RMS error										Means
XOA	2.85	2.30	3.56	3.34	3.68	4.30	2.24	5.37	2.38	4.40	3.44
no selection	29.40	6.26	30.20	30.17	30.18	8.03	30.07	30.17	19.17	30.22	24.39
2XO	3.53	3.47	1.60	4.27	5.37	2.43	3.81	5.37	5.37	2.72	3.79
no selection	8.60	12.59	7.66	33.32	14.40	19.62	6.23	17.37	29.85	23.63	17.33
HCX	4.03	4.04	3.64	4.06	3.93	3.61	3.73	3.20	3.78	3.94	3.80
no selection	9.95	6.32	9.95	11.71	16.59	15.83	7.92	7.37	10.71	8.60	10.49

4 Experimental Results

Three pairs of two groups of ten independent runs were made. In the first pair GPengine crossover (i.e. with append, XOA) was used. In the second pair two-point crossover (without append, 2XO) was used. In the second of each pair selection was turned off by deciding which individuals win or loose each tournament entirely at random. Finally, the last pair used headless chicken crossover (HCX). All runs use point mutation.

In all 3×10 runs with selection, fitness improved and for many generations large parts of the population had the same fitness. Figure 5 shows the evolution of prediction error for the first of ten runs (the others are similar). Cf. Table 2.

Figure 6 shows the evolution of program size. Initially programs are between one and fifteen instructions long, with a mean of seven. However, in runs with fitness selection and crossover (XOA and 2XO) length quickly increases and the longest program is either 500 or very near to this limit. Such bloat was expected [21]. As predicted in Section 3.4, in mutation only runs (HCX) with selection the increase in size is less dramatic. However it was a surprise to see bloat in runs without selection when using crossover with append (XOA) [20]. An initial thought was that this was due to the asymmetric append variation of the crossover operation. This appears to be correct, since when the variant is removed and normal two-point crossover linear GP [6] is used instead, bloat does not appear without fitness selection. (See lower lines in Figure 6.)
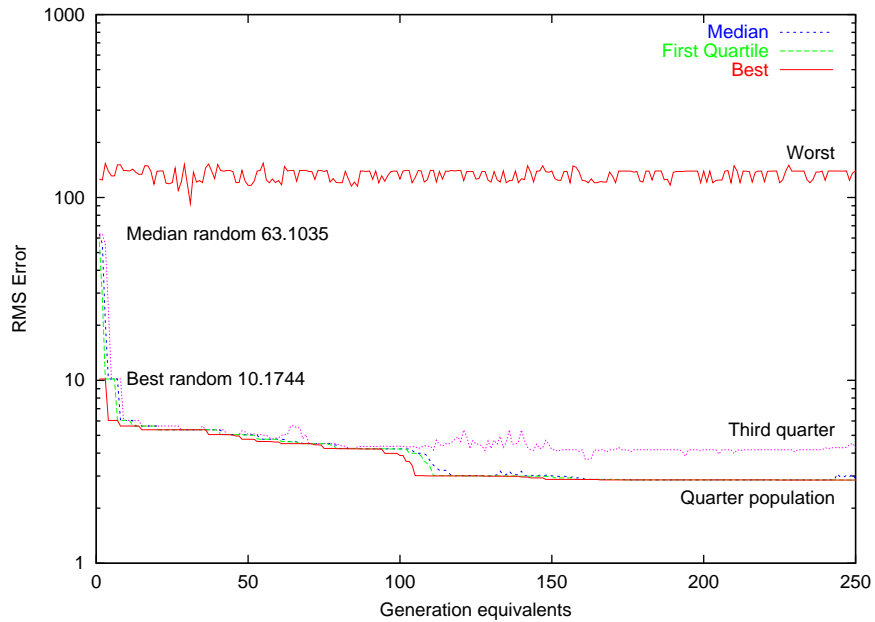
Fig. 5. Evolution of Mackey Glass prediction error (first of ten runs). Note the population chases after the best (lowest) fitness. For many generations at least 25% of the population have the same best fitness. (Sometimes more than half)
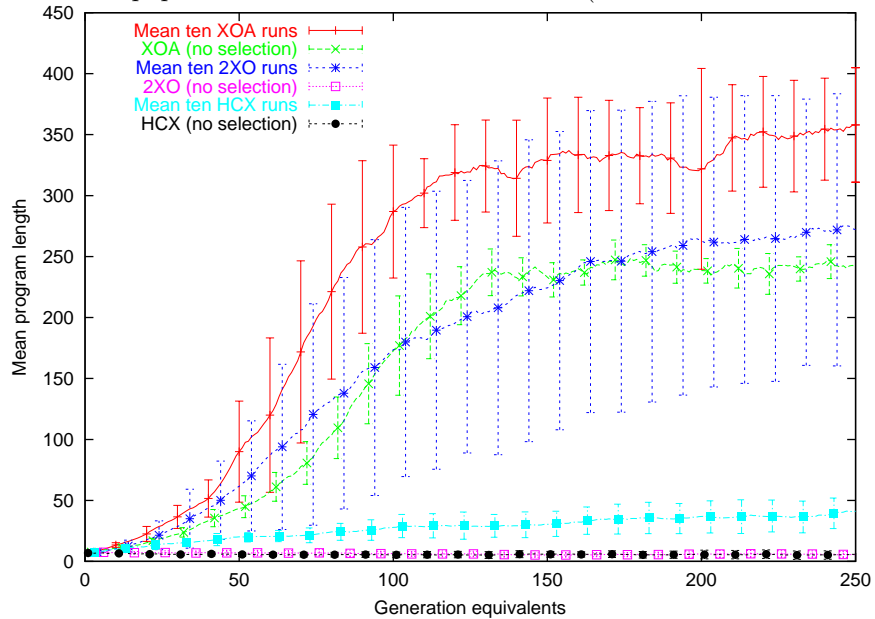

Fig. 6. Evolution of mean program size and variation between runs. Except for the two-point crossover with append (XOA) runs, selection is required for bloat. With crossover (XOA or 2XO) the mean population size appears to increase exponentially, until constrained by the maximum size limit (500).

4.1 Fourier Analysis

Fourier analysis has been applied to DNA sequences. E.g., [12] proposed using the FFT for sequence alignment. But when we calculated the power spectra of a number of programs which contained many repeated sequences, the spectra were similar to those of totally random programs. Only a few frequencies rise above the maximum noise level. Further investigation, perhaps on longer programs, is needed.

4.2 Repeated Program Instruction Sequences

In the random initial programs there are no repeated sequences. They are overwelmingly unlikley to arrive by chance. However, as crossover, mutation and selection get to work and programs grow, instructions start to become repeated. Initially just single instructions are repeated but the length and number of repeats increases (see Figure 7).

All the best programs in the final population of the ten runs with the append crossover variant (XOA) contained repeated sequences (see Figure 8). The longest sequences contained from 12 to 62 instructions. All of these occurred twice, however the programs also contained other, distinct, shorter sequences which occurred multiple times. Again the XOA runs without selection throw up a surprise: eight of the ten best programs[3] contain sequences of instructions which are repeated. All ten runs with two-point crossover without the append variation (2XO), produce repeated sequences, however none of the ten 2XO runs without fitness selection produced repeated instructions.

Figure 8 plots the variation of maximum length of repeated instructions in each of the 4×10 best of run programs against their size. As alternative to saying that repetitive sequences are due to the crossover operator, Figure 8 suggests that the length of the programs (i.e. bloat) is more important. To some extent this is born out by the runs with headless chicken crossover. The ten runs with selection produced best predictors of between 18 and 76 instructions (those without selection contained 2–21 instructions). None contained repeated instructions.

4.3 Effective Code

Rapid increase in length is characteristic of bloat [21]. Analysis (using [9, Algorithm 2]) of the best predictors at the end of XO runs indicates the vast majority of instructions have no impact on the output of the programs. I.e they are ineffective code (introns). Figure 9 shows the distribution of instructions which could affect the prediction along the length of one program. (The other bloated best of 2XO runs are similar but in three runs the best predictors are much shorter and contain only one effective instruction, which is near their end.) There is no obvious correlation between whether an instruction is effective and how many times it is repeated.

---

[3] Even in the absence of selection one can observe quality of programs by evaluating the fitness function.
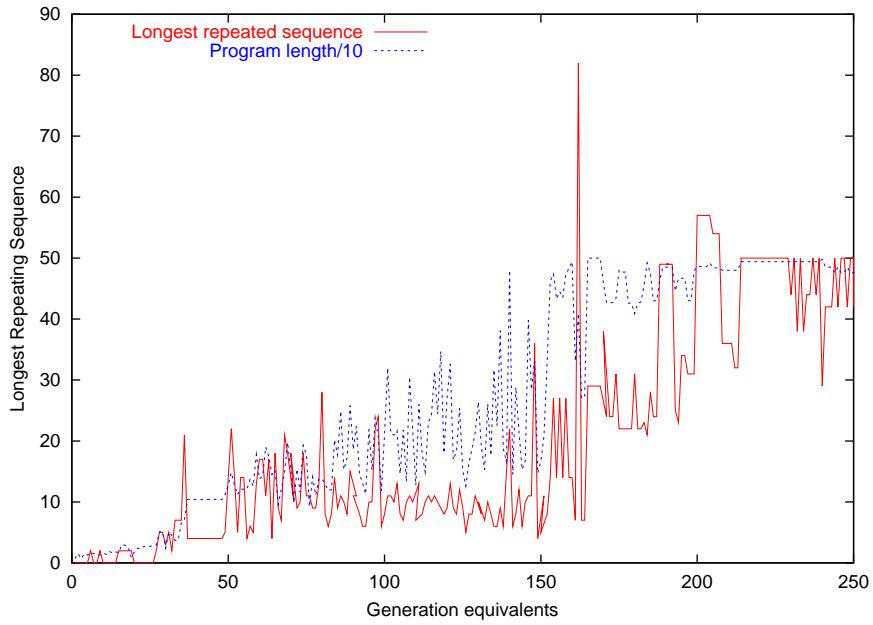
Fig. 7. Evolution of length of longest repeated sequence of instructions in the best Mackey Glass prediction program produced by first run with two-point crossover (2XO) and fitness selection. The length of the programs is also shown.
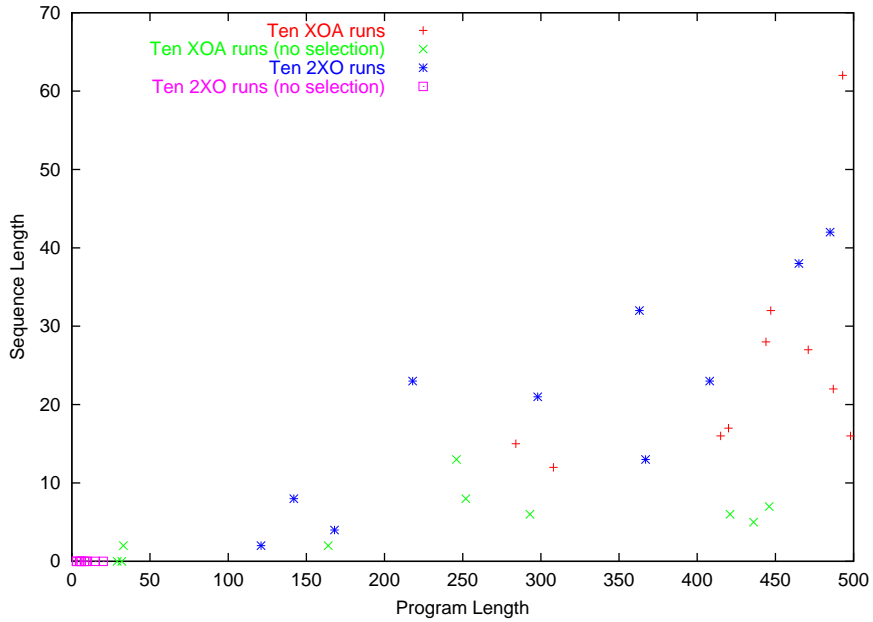

Fig. 8. Length of longest repeated sequence of instructions in the best Mackey Glass prediction program produced by 4×10 runs. With fitness selection both types of crossover evolved repeating sequences. As do 8 of 10 XOA runs without fitness selection but no 2XO runs do when tournaments are random.
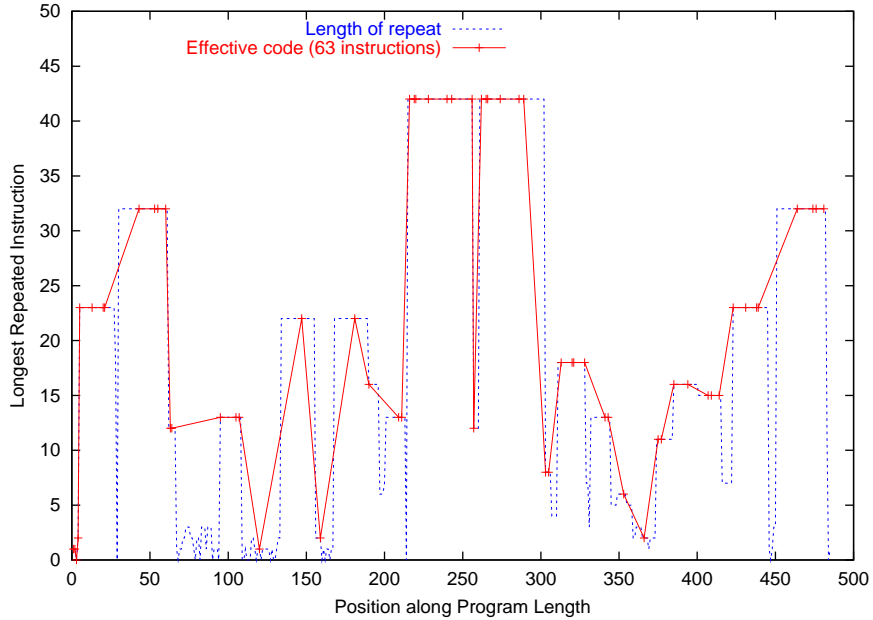
9

Fig. 9. Distribution of repeated sequences along length of best predictor at end of first XO run. It achieves an RMS error 3.53406 using only data from 1 and 8 previous time steps. The solid line highlights the location of its 63 effective instructions. Click here for animations.

4.4 Entropy and Information Content

There are $8 \times 8 \times 4 \times (128 + 8) = 34816$ legal instructions (cf. Figure 1) Since $\log_2 34816 = 15.087463$, a randomly chosen instruction contains slightly more than 15 bits of information. Using this measure suggests that as the population bloats each predictor contains more information. A crude way of estimating actual information content is to compress the programs using gzip [17]. Figure 10 shows information content increase over time but as programs contain more repeated sequences, gzip's Lempel-Ziv algorithm is able to compress the programs, yielding a lower estimate of information content (i.e. higher entropy). Figure 10 shows that gzip (with default parameters and a simple ASCII text format) initially imposes an overhead of about 100 bytes. After about generation 150, gzip is able to recognise patterns in the programs and use them to compress it. For comparison our Mackey-Glass benchmark, without compression, contains 8576 bits (1072 bytes) of information ($1201 \times \log_2 141 = 8576$). I.e. Kolmogorov compression is possible.

5 Conclusion

We have observed the evolution of long repeated sequences of instructions. The chances of them being found purely at random are infinitesimal. However, while we anticipate these sequences to occur widely, so far we have only observed them in one problem domain. We plan to investigate other examples. Of course it is
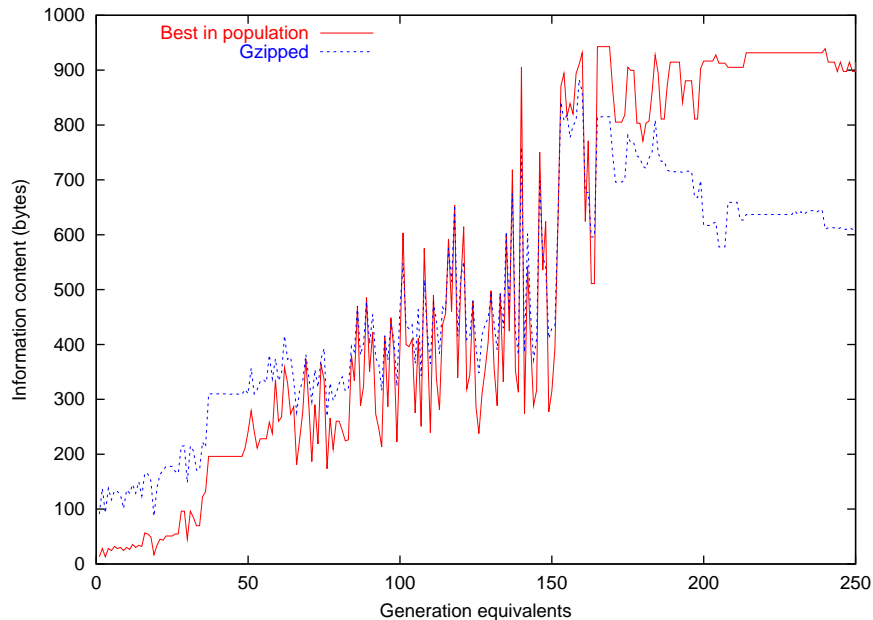
Fig. 10. Evolution of information content in the best Mackey Glass prediction program produced by first run with two-point crossover (2XO) and fitness selection (as measured by gzip). For comparison the length of the programs are also plotted (but normalised so as to give their pre-compressed information content).

interesting to see whether the same happens in tree GP. Most importantly, can these observations be used to help us build better systems in the future? Finally, could experiments of this type in artificial evolution give insight for Biologists concerned about natural evolution?

Acknowledgments

References

1. G. Achaz, E. P. C. Rocha, P. Netter, and E. Coissac. Origin and fate of repeats in bacteria. *Nucleic Acids Research*, 30:2987–2994, 2002.
2. L. Altenberg. Emergent phenomena in genetic programming. In A. V. Sebald and L. J. Fogel, eds., *Evolutionary Programming*, pp 233–241, 1994. World Scientific.
3. P. J. Angeline. Subtree crossover: Building block engine or macromutation? In J. R. Koza, *et al.*, eds., *Genetic Programming 1997*, pp 9–17. Morgan Kaufmann.
4. P. J. Angeline. Subtree crossover causes bloat. In J. R. Koza, *et al.*, eds., *Genetic Programming 1998*, pp 745–752. Morgan Kaufmann.
5. W. Banzhaf and W. B. Langdon. Some considerations on the reason for bloat. *Genetic Programming and Evolvable Machines*, 3(1):81–91, Mar. 2002.

6. W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming – An Introduction.* Morgan Kaufmann, 1998.

7. G. Benson. Tandem repeats finder: A program to analyze DNA sequences. *Nucleic Acids Research*, 27:573–580, 1999.

8. J. W. Bizzaro and K. A. Marx. Poly: a quantitative analysis tool for simple sequence repeat (SSR) tracts in DNA. *BMC Bioinformatics*, 4:22–28, 2003.

9. M. Brameier and W. Banzhaf. A comparison of linear genetic programming and neural networks in medical data mining. *IEEE Trans. EC*, 5(1):17–26, Feb. 2001.

10. R. J. Britten and D. E. Kohnen. Repeated sequences in DNA. *Science*, 161:529–540, 1968.

11. O. Elemento, O. Gascuel, and M. P. Lefranc. Reconstructing the duplication history of tandemly repeated genes. *Molec Biology and Evolution*, 19:278–288, 2002.

12. J. Felsenstein, S. Sawyer, and R. Kochin. An efficient method for matching mucleic acid sequences. *Nucleic Acid Research*, 10(1):133–139, 1982.

13. F. D. Francone, M. Conrads, W. Banzhaf, and P. Nordin. Homologous crossover in genetic programming. In W. Banzhaf, *et al.*, eds., *GECCO*, pp 1021–1026, Orlando, 13-17 July 1999. Morgan Kaufmann.

14. J. V. Hansen. Genetic programming experiments with standard and homologous crossover methods. *GP and Evolvable Machines*, 4(1):53–66, Mar. 2003.

15. L. C. Hsieh and H. C. Lee. Model for the growth of bacterial genomes. *Modern Physics Letters*, 16:821–827, 2002.

16. Z. Izsvak, Z. Ivics, and P. B. Hackett. Repetitive elements and their applications in zebrafish. *Biochemical Cell Biology*, 75:507–523, 1997.

17. W. B. Langdon. *Genetic Programming and Data Structures.* Kluwer, 1998.

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

19. W. B. Langdon. Convergence rates for the distribution of program outputs. In W. B. Langdon, *et al.*, eds., *GECCO 2002*, pp 812–819. Morgan Kaufmann.

20. W. B. Langdon and R. Poli. *Foundations of GP.* Springer, 2002.

21. W. B. Langdon, T. Soule, R. Poli, and J. A. Foster. The evolution of size and shape. In L. Spector, *et al.*, eds., *Advances in GP 3*, pp 163–190. MIT Press, 1999.

22. J. R. Lupski and G. M. Weinstock. Short, interspersed repetitive DNA sequences in procaryotic genomes. *Journal of Bacteriology*, 174:4525–4529, 1992.

23. P. Nordin, F. Francone, and W. Banzhaf. Explicitly defined introns and destructive crossover in GP. In P. J. Angeline *et al.*, eds., *Advances in GP 2*, pp 111–134. 1996.

24. H. Oakley. Two scientific applications of genetic programming. In K. E. Kinnear, Jr., ed., *Advances in GP*, pp 369–389. MIT Press, 1994.

25. C. Patience, D. A. Wilkinson, and R. A. Weiss. Our retroviral heritage. *Trends in Genetics*, 13:116–120, 1997.

26. A. Singleton. Walter Tackket's PhD thesis citing Andy Singleton "personal communication" as proposing the "intron" explanination for bloat in GP trees., 1994.

27. A. F. A. Smit. The origin of interspersed repeats in the human genome. *Current Opinions in Genetics and Development*, 6:743–748, 1996.

28. G. P. Smith. Evolution of repeated DNA sequences by unequal crossover. *Science*, 191(4227):528–535, 13 Feb 1976.

29. W. A. Tackett. *Recombination, Selection, and the Genetic Construction of Computer Programs.* PhD thesis, EES, University of Southern California, 1994.

30. A. Taneda. Adplot: detection and visulization of repetitive patterns in complete genomes. *Bioinformatics*, 5:701–708, 2004.

31. G. Toth, Z. Gaspari, and J. Jurka. Microsatellites in different eukaryotic genomes: Survey and analysis. *Genome Research*, 10:967–981, 2000.