Preprint Revision : 1.30 Complex Systems 2005 15 (4) pp285-306

Repeated Sequences in Linear Genetic Programming Genomes

William B. Langdon

Computer Science, University College, London, Gower Street, London, UK

Wolfgang Banzhaf

Computer Science, Memorial University of Newfoundland, St.John's, A1B 3X5, CANADA

Biological chromosomes are replete with repetitive sequences, micro satellites, SSR tracts, ALU, etc. in their DNA base sequences. We started looking for similar phenomena in evolutionary computation. First studies find copious repeated sequences, which can be hierarchically decomposed into shorter sequences, in programs evolved using both homologous and two point crossover but not with headless chicken crossover or other mutations. In bloated programs the small number of effective or expressed instructions appear in both repeated and non-repeated code. Hinting that building-blocks or code reuse may evolve in unplanned ways.

Mackey-Glass chaotic time series prediction and eukaryotic protein localisation (both previously used as artificial intelligence machine learning benchmarks) demonstrate evolution of Shannon information (entropy) and lead to models capable of lossy Kolmogorov compression. Our findings with diverse benchmarks and GP systems suggest this emergent phenomenon may be widespread in genetic systems.

"DNA whose sequence is not maintained by selection will develop periodicities as a result of random crossover" George P Smith [36].

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 [34, 38, 2]. It was also noted that bloat affects many GP systems. Reasons for bloat and non-effective code have been examined in years past [25, 4, 7] and remedies have been developed more or less effective under particular circumstances (e.g. [29, 15, 22, 17]).

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. This is interesting in itself and 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 two linear GP system used for our experiments and the time series prediction and Bioinformatics protein classification tasks they were applied to. Section 4 presents results of our experiments. Section 5 concludes.

2. Biological Background

Biologists have discovered that there is a vast amount of repetition in the DNA of microbes, plants and animals [12]. Less than 3% of a human genome consists of protein-coding genes but about 50% of it consists of repetitive sequences, many of viral origin [35, 31]. Initially Biologists concentrated upon understanding the protein-coding part of genomes. However with whole genome analysis becoming more common, repetitive DNA is a lively subject of research [26, 40, 1].

There are many forms of repeated DNA. This multitude confirms that it is a complex phenomenon. There are satellites, mini-satellites and micro-satellites. Repeats of different sizes are located next to each other along the genome. There are ALU repeats and interspersed repetitive sequences. Repeating sequences are found in coding, non-coding and intercistronic areas. Repeats are distributed over genomes and species and constitute a considerable fraction of all DNA in many organisms.

The search for causes began some time ago. Smith, in 1976, did numerical experiments in order to explain evolution of repeated DNA sequences [36]. 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 to play an important role. Driven by the inaccuracy of the DNA replication machinery, repeated sequences are both a consequence of misalignments and a cause for crossover [13]. Hsieh and Lee considered a model of bacterial genome growth working with a mechanism called "random segmental self-copying" [18]. This model was able to explain, at a statistical level, the distribution of patterns found in bacterial genomes. They concluded, since the statistical traces are still visible in the distribution of DNA patterns, that growth processes of genomes must have taken place.

In recent years, quantitative analysis tools have become available in molecular biology that allow a closer look at these phenomena [9, 10, 39]. These will provide the opportunity to observe even more closely how different repetitive patterns emerged during evolution. Also applications of repetitive sequences are starting to appear [19].

3. The Linear Genetic Programming System

Genetic programming (GP) has been described many times [20, 8, 21]. Essentially GP applies the well known genetic algorithm (GA) to the task of searching for a program which does what is needed. Unlike the typical GA, GP evolves both the numeric values and the form of the solution. This additional freedom allows trial solutions of changing complexity and opens up evolutionary computation to new, unexpected and emergent effects.

We decided to start with linear genetic programming [27, 5, 6, 32] in which the evolved data or chromosome is a linear program. Such a linear structure is analogous to the DNA molecule in biological genomes. This makes linear GP a good place to start looking. A linear structure is also intrinsically easier to search for repeated substructures.

In order to show the wide-spread nature of repeated sequences, we used two radically different linear genetic programming systems. GPengine, a standard academic C++ system, and a commercial machine code GP system, Discipulus. To allow repeatability, Sections 3.1 to 3.7 describe GPengine and Discipulus in detail. While Sections 3.8 and 3.9 describe the two benchmarks used. Sections 4.1 and 4.2 will describe the results obtained, while the remainder of Section 4 will describe the repeated sequences evolved by crossover.

3.1 Tournament Selection and Steady State Population

GPengine and Discipulus both use a steady state population and tournament selection. In a steady state population individuals are steadily added and removed from the population [37]. In contrast, in a generational system, at the end of each generation, the whole population is replaced by the next. In steady state systems there are no distinct generations. In terms of measuring evolutionary time in a population of M individuals, a generation equivalent is the time taken to create and kill M individuals.

GPengine and Discipulus tournament selection are similar. 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 that they have identical fitness (e.g. 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.

The offspring produced from the two winners (by crossover, mutation or copying (cloning)) replace the two losers. Note, that 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. The best individual, however, 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).

Discipulus differs only in details. Discipulus saves the best program found so far as it runs. At the end of a run, we analyse the best of these. Note that this program may have been found hundreds of generations before the end of the run. While in GPengine runs, when we say the best program, we will mean the best program in the population at the end of the run.

3.2 Linear GP Representation and Program Evaluation

In GPengine each individual consists of a linear sequence of instructions. After input to the program is provided by initializing certain registers, the sequence is executed and register values are changed accordingly. By convention, the output of a GP program resides in the first register (R0).

Each instruction takes two inputs, performs its (integer) 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. As mentioned, 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.

[1] GPengine uses the C rand function.

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

Figure 1. Format of a GPengine instruction.

Again, Discipulus is slightly different. It does not interpret the instructions, it executes them directly. To allow this they are Intel 486 machine code instructions, packed into 4 byte words (padded with nops where needed to fill 4 bytes). Inputs and constants are held in separate registers from the read/write floating point arithmetic registers.

3.3 GPengine Crossover's (XOA and 2XO)

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 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 loser 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 Discipulus Crossovers (2XO and HXO)

Discipulus provides two crossover operators. The first, two point, is essentially the same as GPengine's two point crossover. The second is called "homologous" crossover (HXO) (see Figure 4). Here the same two crossover points are used in both parents. So while HXO exchanges code, the offspring are the same length as their parents [28].

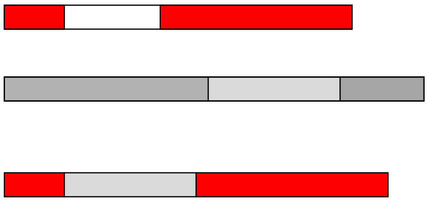

Figure 2. GPengine crossover. Two instructions are randomly chosen in each parent (top two genomes) as cut points. If the code to be inserted from the second parent at least partially overlaps the first, it is inserted in the normal way to give the child (lower chromosome). With headless chicken crossover, the inserted code is randomly generated.

Figure 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.)

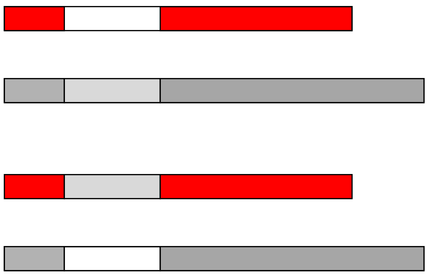

Figure 4. HXO crossover. As with the other crossover operations, two parents (top two programs) crossover to yield two child programs (bottom). In HXO the two crossover cut points are the same in both parents. Note code does not change its position relative to the start of the program (left edge) and the child programs are the same lengths as their parents.

3.5 Variable length Mutation – Headless Chicken Crossover

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 introduced 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, unlike Discipulus, does not write-protect its inputs, this means that 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 [23]. 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.

3.6 GPengine Point 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. Point 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.

3.7 Discipulus Mutation

Discipulus mutation is similar to GPengine's point mutation. It, too, mutates offspring whether they have been created by crossover or copied (reproduced) directly from their parents. There are three types of Discipulus mutation. Each ensures that offspring is still a valid machine code program. They do not change the program's size. They are 1) replace a 4 byte instruction block with another 4 bytes of randomly generated machine code, 2) replace an instruction (op-code) with another machine code instruction of the same length, 3) replace an instructions operand (input, variable or constant) with another input variable or constant.

3.8 Mackey-Glass Benchmark and GPengine

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 [30]. 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 5.

The task for the GP is, given historical data, to predict the next value. Since the series is chaotic this cannot be done exactly. GP is given eight earlier values from 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.

3.9 Locating Animal Proteins with Discipulus

Once again we wanted a challenging problem but radically different from the predicting tomorrow's share price from a time series. So we chose a binary classification (rather than continuous regression) problem from biology (rather than finance or engineering). It is possible to predict the function of a protein from the fractions of each amino acid from which it is made. This has the advantage that, in many cases, data is readily available from Swissprot or can also be inferred from gene

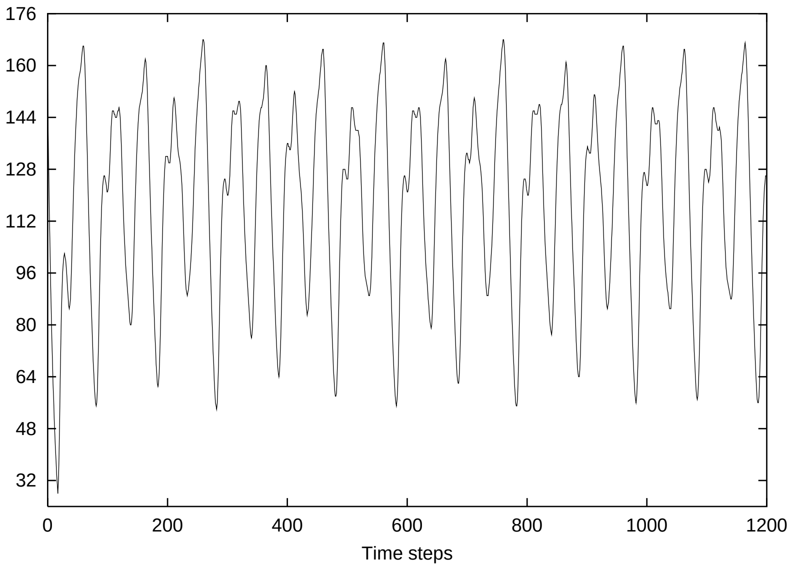

Figure 5. Discrete Mackey-Glass chaotic time series

DNA sequences. Naturally, with such a crude measure only limited predictions can be made. Nevertheless Reinhardt and Hubbard [33] showed anino acid composition can be used to predict the location of the protein from which it may be possible to infer something of its function. In [33] they used machine learning to differentiate between seven cellular locations spread across both animals (eukaryotic, they excluded plants) and microbes (prokaryotic).

We restrict ourselves to localising animal proteins (one normally knows if a protein is animal or bacterial). Since we are seeking a binary classification problem, we evolve models which predict if an animal protein will be found in the cell nucleus or elsewhere. (I.e. in the cell cytoplasm, in the mitochondria or outside the cell [33].) We used the same Swissprot data for 2427 proteins as [33]. There are 1097 nuclear (and 1330 non-nuclear) sequences of amino acids (see Figure 6).

We counted the number of each amino acid in each protein and allocated nuclear proteins to class 1 and the others to class 0. This gave 2427 records, each containing 20 integers and a class label (0/1). Following [14] we split these evenly into a training and a test file.

4. Experimental Results

4.1 Predicting the Mackey-Glass chaotic time series

Three pairs of two groups of ten independent runs were made. In the first pair GPengine's default crossover (i.e. with append, XOA) was

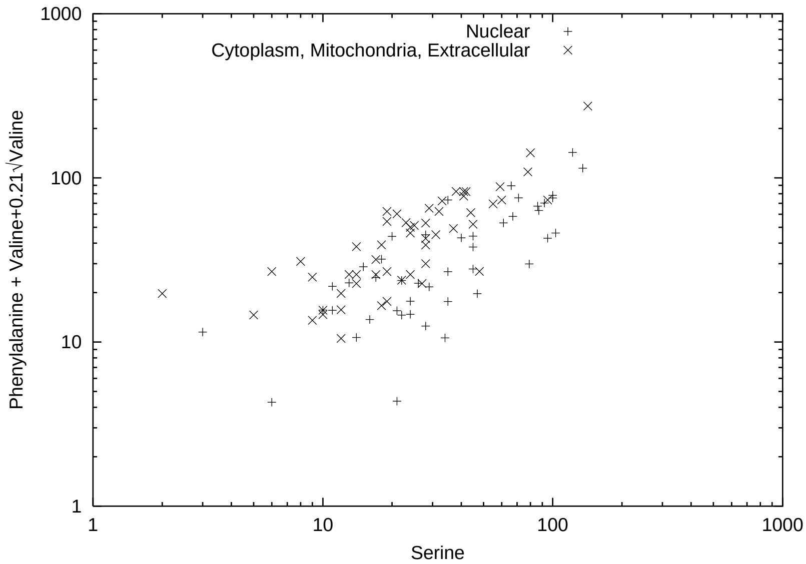

Figure 6. Number of amino acids in Nuclear and non-nuclear proteins used with Discipulus. To reduce clutter only 5% of the data are presented. The three (of twenty) amino acids plotted where selected by Discipulus as being the best discriminators. The non-linear function of number of Valines was also suggested by an evolved model.

used. In the second pair two-point crossover (without append, 2XO) was used. Finally, the last pair used headless chicken crossover (HCX). In the second of each pair, selection was turned off by deciding which individuals win or loose each tournament entirely at random. All runs use point mutation (cf. Table 1).

In all 3×10 runs with selection, fitness improved and for many generations large parts of the population had the same fitness. Figure 7 shows the evolution of prediction error for the first of ten runs (the others are similar). Cf. Table 2.

Figure 8 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 [25]. As predicted in Section 3.5, 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) [24]. 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 is used instead, bloat does not appear without fitness selection. (See

Objective:	Evolve a prediction for a chaotic time series
Function set:	$+\ -\ \times\ \div^{a}$ (operating on unsigned bytes)
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:	125 500 individuals evaluated

[a] If second argument of $\div$ is zero, $\div$ returns zero.

Table 1. GPengine parameters for Mackey-Glass time series prediction.

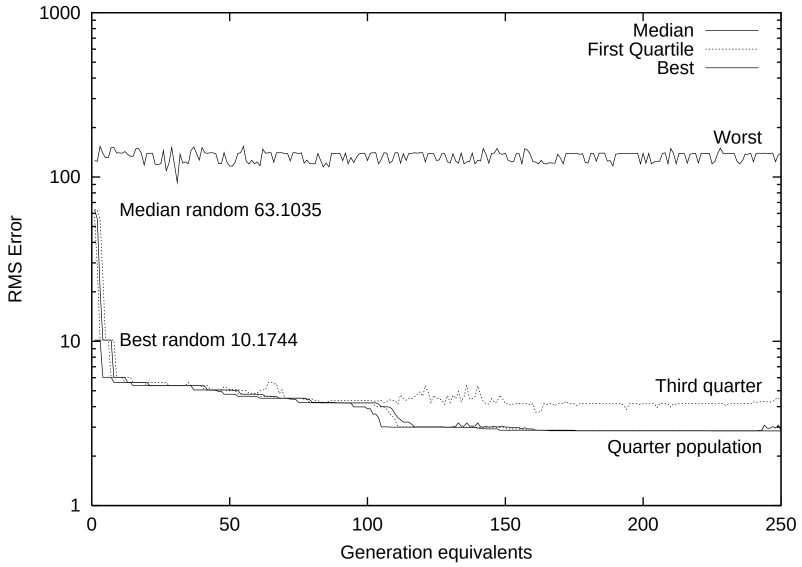

Figure 7. Evolution of Mackey-Glass prediction error (first of ten runs). Note the population chases after the best (lowest) fitness. For many generations at least 25% of the population have the same best fitness. (Sometimes more than half)

	RMS error										Mean
XOA	2.85	2.30	3.56	3.34	3.68	4.30	2.24	5.37	2.38	4.40	3.44
	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
	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
	9.95	6.32	9.95	11.71	16.59	15.83	7.92	7.37	10.71	8.60	10.49

Table 2. Best Mackey-Glass prediction error at end of GPengine runs. Using two point crossover with append (XOA), without append (2XO) and headless chicken crossover crossover (HCX). Lower row of each pair refers to the runs without fitness selection.

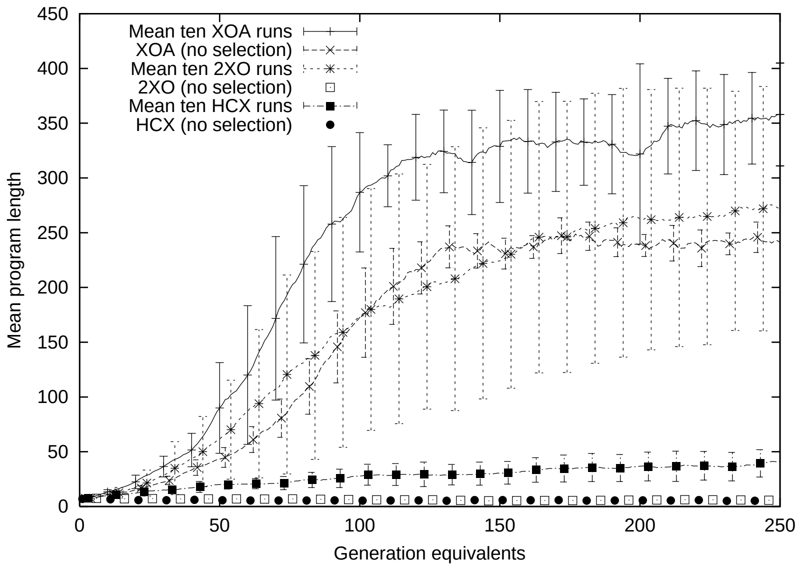

Figure 8. Evolution of mean program size and variation between Mackey-Glass prediction runs. (Variation between HCX and 2XO without selection, lowest plots, is too small to show, and so is omitted.) Except for the two-point crossover with append (XOA) runs, selection is required for bloat. With crossover (XOA or 2XO) the mean population size appears to increase exponentially, until constrained by the maximum size limit (500).

lower lines in Figure 8.)

4.2 Predicting Protein Location

For each experiment on the Swissprot animal proteins (cf. Section 3.9) Discipulus was run ten times, always in classification mode and with identical parameters but with different initial random number seeds. As far as possible we used Discipulus default configurations (actual val-

Objective:	Evolve a prediction of nuclear or non-nuclear location for animal proteins based on their amino acid composition
Terminal set:	2 read-write FPU registers, 43 randomly chosen constants. Number (integer) of each of the 20 amino acids in the protein. (Codes B and Z are ambiguous. Counts for B were split evenly between aspartic acid D and asparagine N. Those for Z between glutamic acid E and glutamine Q.)
Fitness:	DSS [16, 14]. Parsimony not used.
Selection:	Steady state, tournament 2 by 2
Initial pop:	Random program's length uniform chosen from 4..80 bytes
Parameters:	Population 500 (10×50 demes), max program size 2048 (bytes), 95% crossover (either all 2XO or 95% HCX and 5% 2XO) 95% mutation (three types 30%, 30%, 40%)
Termination:	500 000 individuals evaluated

Table 3. Discipulus parameters used in animal protein location prediction experiments. Only the maxiumnum program size and HCX were changed from factory defaults.

Percent correct Nuclear vs. non-Nuclear prediction											Mean
2XO	82	80	80	80	80	81	82	78	79	81	80.3
HXO	81	81	79	80	80	79	78	81	78	81	80.0

Table 4. Accuracy of best animal protein location predictors evolved using Discipulus two point (2XO) and HXO (homologous) crossover. Performance on training and test datasets is always within a few percent of each other (indicating little over fitting) therefore we report performance averaged across both.

ues are given in Table 3). In both experiments the maximum program size was 2048 bytes. In the first experiment, "homologous crossover" was disabled, whilst in the second it was left at its default rate (95%). Table 4 gives the accuracy of the best models found by each run.

As an aside, when Discipulus was allowed to run in its default multirun mode, it evolved a team solution with an accuracy of 87.1%. Reinhardt and Hubbard [33] did not do a nuclear vs. non-nuclear classification. Instead they report results for the easier problems of classifying animal nuclear proteins against each of their other classes one at a time. Table 2 in [33] gives mean accuracies of 84.9%, 84.8% and 86.1% when classifying nuclear proteins against each of the three locations individually. Given the variance of the neural network runs, one cannot say that Discipulus did significantly better, but it is doing at least as well on a harder version of the problem.

4.3 Repeated Program Instruction Sequences

In the random initial GPengine programs there are no repeated sequences. They are overwelmingly unlikely 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 9).

All the best programs of the ten runs with the append crossover variant (XOA) contained repeated sequences (see Figure 10). 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[2] 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.

Both the ten 2XO and ten HXO Discipulus runs, evolved binary classification programs with repeated instructions like those found in GPengine 2XO runs. In 19 cases the longest sequence was between 6 and 64 instructions. (The remaining run found a short program containing a few individual instructions repeated a small number of times, but no repeated sequences.)

Figure 10 plots the variation of maximum length of repeated instructions in each of the 6×10 best of run programs against their size. As alternative to saying that repetitive sequences are due to the crossover operator, Figure 10 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.

Figure 11 shows the location of repeated sequences in a single evolved program. Figure 11 stresses the shear number of repeated instructions. It also shows some repeated instructions are part of long sequences (up to 10% of program length) and that repeated instructions appear throughout the program. (The tartan pattern in Figure 11 suggests their may be other structures which have yet to be investigated.)

Our results strongly suggests that crossover is responsible for re-

[2]Even in the absence of selection one can observe the quality of programs by evaluating the fitness function.

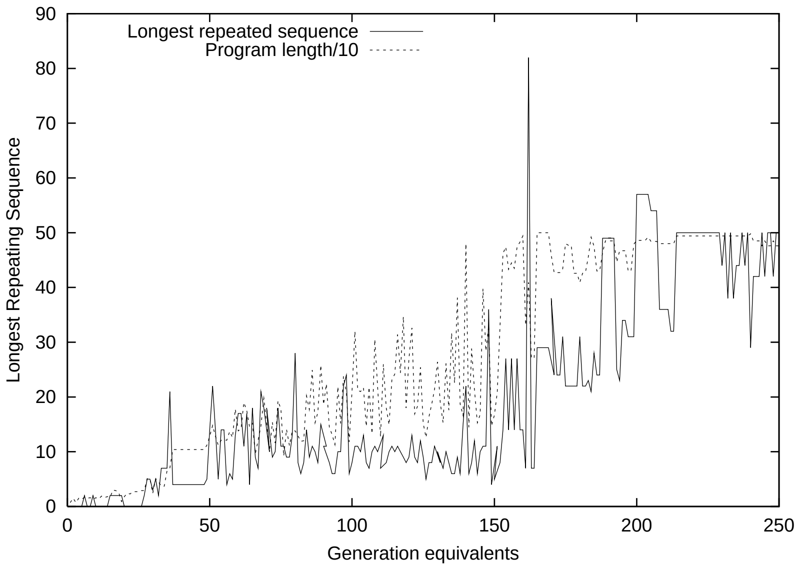

Figure 9. Evolution of length of longest repeated sequence of instructions in the best Mackey-Glass prediction program produced by the first run with two-point crossover (2XO) and fitness selection. The length of the programs is also shown.

peated sequences. However we cannot rule out the possibility that bloated long programs produced by mutation might also contain repeated sequences, although this does seem unlikely. As mentioned before, mutation is unable to produce long programs, so we cannot test this.

4.4 Effective Code

Rapid increase in length is characteristic of bloat [25]. We used algorithm 2 from [11] to analyse the best predictors of the GPengine 2XO and XOA runs. We also used Discipulus' own intron removal tool. These showed that the majority of instructions have no impact on the output of the programs. I.e., they are ineffective code (introns). Figure 12 shows the distribution of instructions which could affect the prediction along the length of one program. The other bloated best of 2XO GPengine runs are similar to that shown in Figure 12. However in three runs there was less bloat. Their best predictors are much shorter. They contain only one effective instruction, which is near the end. There is no obvious correlation between whether an instruction is effective and how many times it is repeated.

As the fraction of ineffective code increases so mutation is more and more unlikely to change a program's performance. This is (part of)

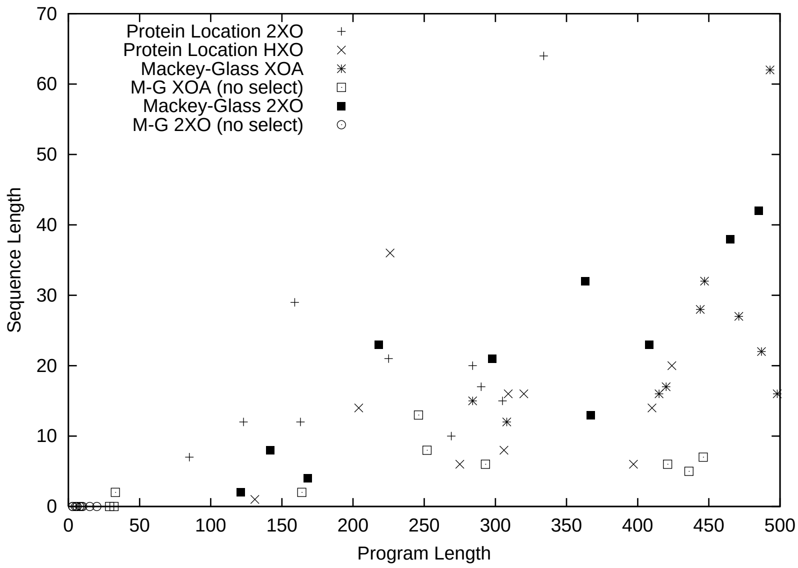

Figure 10. Length of longest repeated sequence of instructions in the best prediction programs. 2×10 Discipulus protien locations runs and 4×10 Mackey-Glass prediction runs. With fitness selection all three types of crossover evolved repeating sequences. As do 8 of 10 XOA runs without fitness selection but no 2XO runs do when tournaments are random.

the reason why Mackey-Glass populations converge towards a single fitness, cf. Figure 7. Similarly as evolution increases the number of repeated instructions, mutation increasingly often duplicates an existing instruction. However, on average, 88% of point mutations to the best Mackey-Glass programs evolved by 2XO produce a new instruction.

4.5 Entropy and Information Content

There are $8 \times 8 \times 4 \times (128 + 8) = 34816$ GPengine 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 [21]. Figure 13 shows information content increases over time but as programs contain more repeated sequences, gzip's Lempel-Ziv algorithm is able to compress the programs. Smaller size gives a lower estimate of information content of the programs. Figure 13 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 compari-

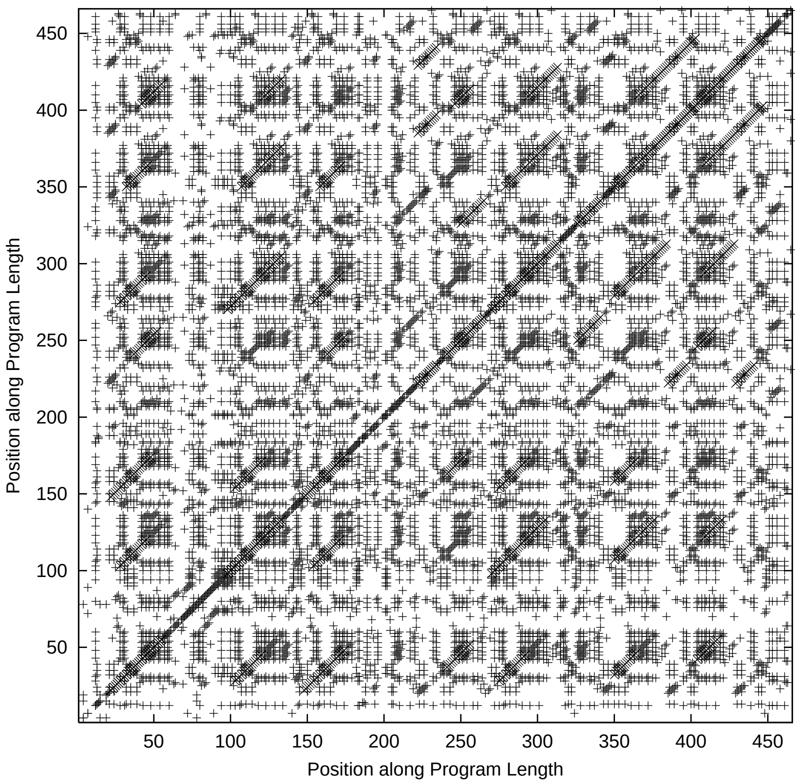

Figure 11. Location of repeated instructions in the best Mackey-Glass prediction program produced by third run with two-point crossover (2XO) and fitness selection. Instructions that are part of repeated sequences longer than 10 are plotted with $\times$, those less, with $+$. This run chosen as it best highlights the structure. Notice almost every instruction is repeated, so the diagonal is almost solid. (The figure is symmetric about the diagonal.) Each cross in a vertical line, say x=400, indicates an instruction which is identical (to that at position 400).

son our Mackey-Glass benchmark, without compression, contains 8576 bits (1072 bytes) of information ($1201 \times \log_2 141 = 8576$). Lossy (programmatic) Kolmogorov compression is possible and achieves 7.6 times more compression than gzip's lossless compression.

5. Conclusion

Approximately half of human DNA is composed of repeated sequences. Initial experiments using two very different linear GP systems (one evolving Intel floating point machine code, one using a C++ interpreter of integer arithmetic) on two different benchmarks (one time series pre-

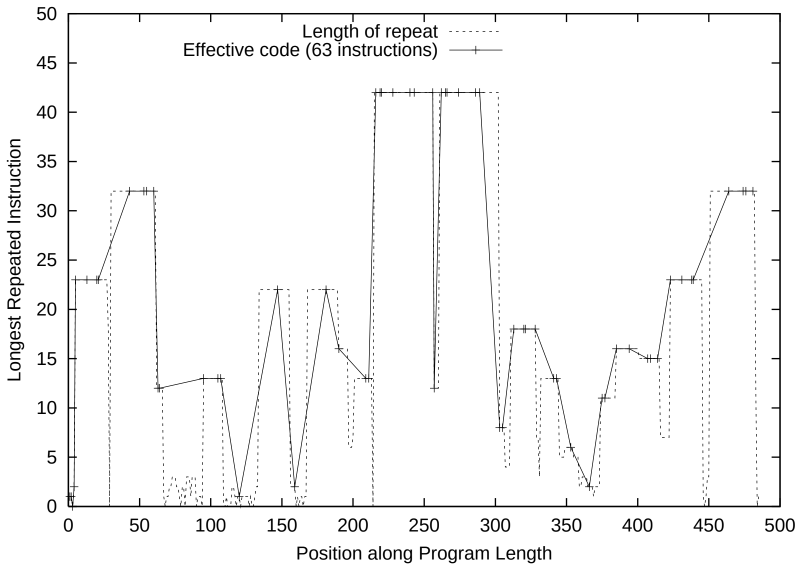

Figure 12. Distribution of repeated sequences along length of best Mackey-Glass predictor at end of first 2XO run. It achieves an RMS error 3.5 using only data from 1 and 8 previous time steps. The solid line highlights the location of its 63 effective instructions. An animation can be found via `http://www.cs.ucl.ac.uk/staff/W.Langdon/gecco2004lb/`

diction, the other a Bioinformatics binary classification) suggest where artificial genetic systems have the space to evolve repeated patterns, they will emerge. In our runs, both the number of repeated sequences and their length increase with time but so, too, does the length of the programs. These bloated programs are not random. Evidence suggests that crossover is responsible.

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 occur widely, so far we have only observed them in linear genetic programming (GP). Of course it is interesting to see whether the same, or other emergent phenomena, occur 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? For example, are statistical distributions of repeated sequences comparable to what happens in real genomes?

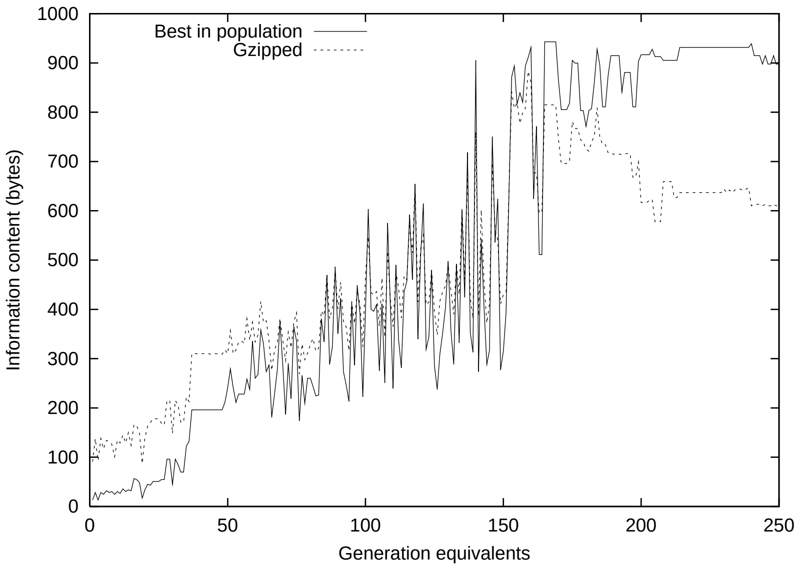

Figure 13. Evolution of information content in the best Mackey-Glass prediction program produced by first run with two-point crossover (2XO) and fitness selection (as measured by gzip). For comparison the length of the programs are also plotted (but normalised so as to give their pre-compressed information content).

Acknowledgments

The linear genetic programming system, GPengine, was given by Peter Nordin. We would like to thank Paul Gillard, Marian Wissink, Nolan White, Dick Furnstahl, Frank D. Francone, Markus Conrads and David H. Jones. Support was provided by a grant from the visitor program of the Department of Computer Science, Memorial University of Newfoundland.

References

[1] Achaz, G., Rocha, E. P. C., Netter, P., and Coissac, E. Origin and fate of repeats in bacteria. *Nucleic Acids Research 30* (2002), 2987–2994.

[2] Altenberg, L. Emergent phenomena in genetic programming. In *Evolutionary Programming — Proceedings of the Third Annual Conference* (San Diego, CA, USA, 24-26 Feb. 1994), A. V. Sebald and L. J. Fogel, Eds., World Scientific Publishing, pp. 233–241.

[3] Angeline, P. J. Subtree crossover: Building block engine or macromutation? In *Genetic Programming 1997: Proceedings of the Second*

Annual Conference (Stanford University, CA, USA, 13-16 July 1997), J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, Eds., Morgan Kaufmann, pp. 9–17.

[4] Angeline, P. J. Subtree crossover causes bloat. In *Genetic Programming 1998: Proceedings of the Third Annual Conference* (University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998), J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo, Eds., Morgan Kaufmann, pp. 745–752.

[5] Atkin, M. S., and Cohen, P. R. Learning monitoring strategies: A difficult genetic programming application. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence* (Orlando, Florida, USA, 27-29 June 1994), IEEE Press, pp. 328–332a.

[6] Banzhaf, W. Genetic programming for pedestrians. In *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93* (University of Illinois at Urbana-Champaign, 17-21 July 1993), S. Forrest, Ed., Morgan Kaufmann, p. 628.

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

[8] Banzhaf, W., Nordin, P., Keller, R. E., and Francone, F. D. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, dpunkt.verlag, Jan. 1998.

[9] Benson, G. Tandem repeats finder: A program to analyze DNA sequences. *Nucleic Acids Research 27* (1999), 573–580.

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

[11] Brameier, M., and Banzhaf, W. A comparison of linear genetic programming and neural networks in medical data mining. *IEEE Transactions on Evolutionary Computation 5*, 1 (Feb. 2001), 17–26.

[12] Britten, R. J., and Kohnen, D. E. Repeated sequences in DNA. *Science 161* (1968), 529–540.

[13] Elemento, O., Gascuel, O., and Lefranc, M. P. Reconstructing the duplication history of tandemly repeated genes. *Molecular Biology and Evolution 19* (2002), 278–288.

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

[15] Francone, F. D., Conrads, M., Banzhaf, W., and Nordin, P. Homologous crossover in genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Orlando, Florida, USA, 13-17 July 1999), W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, Eds., vol. 2, Morgan Kaufmann, pp. 1021–1026.

[16] Gathercole, C. *An Investigation of Supervised Learning in Genetic Programming*. PhD thesis, University of Edinburgh, 1998.

[17] Hansen, J. V. Genetic programming experiments with standard and homologous crossover methods. *Genetic Programming and Evolvable Machines 4*, 1 (Mar. 2003), 53–66.

[18] Hsieh, L. C., and Lee, H. C. Model for the growth of bacterial genomes. *Modern Physics Letters 16* (2002), 821–827.

[19] Izsvak, Z., Ivics, Z., and Hackett, P. B. Repetitive elements and their applications in zebrafish. *Biochemical Cell Biology 75* (1997), 507–523.

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

[21] Langdon, W. B. *Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming!*, vol. 1 of *Genetic Programming*. Kluwer, Boston, 24 Apr. 1998.

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

[23] Langdon, W. B. Convergence rates for the distribution of program outputs. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference* (New York, 9-13 July 2002), W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, Eds., Morgan Kaufmann Publishers, pp. 812–819.

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

[25] Langdon, W. B., Soule, T., Poli, R., and Foster, J. A. The evolution of size and shape. In *Advances in Genetic Programming 3*, L. Spector, W. B. Langdon, U.-M. O'Reilly, and P. J. Angeline, Eds. MIT Press, Cambridge, MA, USA, June 1999, ch. 8, pp. 163–190.

[26] Lupski, J. R., and Weinstock, G. M. Short, interspersed repetitive DNA sequences in prokaryotic genomes. *Journal of Bacteriology 174* (1992), 4525–4529.

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

[28] Nordin, P., Banzhaf, W., and Francone, F. D. Efficient evolution of machine code for CISC architectures using instruction blocks and homologous crossover. In *Advances in Genetic Programming 3*, L. Spector, W. B. Langdon, U.-M. O'Reilly, and P. J. Angeline, Eds. MIT Press, Cambridge, MA, USA, June 1999, ch. 12, pp. 275–299.

[29] Nordin, P., Francone, F., and Banzhaf, W. Explicitly defined introns and destructive crossover in genetic programming. In *Advances in Genetic Programming 2*, P. J. Angeline and K. E. Kinnear, Jr., Eds. MIT Press, Cambridge, MA, USA, 1996, ch. 6, pp. 111–134.

[30] Oakley, H. Two scientific applications of genetic programming: Stack filters and non-linear equation fitting to chaotic data. In *Advances in Genetic Programming*, K. E. Kinnear, Jr., Ed. MIT Press, 1994, ch. 17, pp. 369–389.

[31] Patience, C., Wilkinson, D. A., and Weiss, R. A. Our retroviral heritage. *Trends in Genetics 13* (1997), 116–120.

[32] Perkis, T. Stack-based genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence* (Orlando, Florida, USA, 27-29 June 1994), vol. 1, IEEE Press, pp. 148–153.

[33] Reinhardt, A., and Hubbard, T. Using neural networks for prediction of the subcellular location of proteins. *Nucleic Acids Research 26*, 9 (1 May 1998), 2230–2236.

[34] Singleton, A. Walter Tackett's PhD thesis citing Andy Singleton "personal communication" as proposing the "intron" explanation for bloat in GP trees., 1994.

[35] Smit, A. F. A. The origin of interspersed repeats in the human genome. *Current Opinions in Genetics and Development 6* (1996), 743–748.

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

[37] Syswerda, G. Uniform crossover in genetic algorithms. In *Proceedings of the third international conference on Genetic Algorithms* (George Mason University, 4-7 June 1989), J. D. Schaffer, Ed., Morgan Kaufmann, pp. 2–9.

[38] Tackett, W. A. *Recombination, Selection, and the Genetic Construction of Computer Programs*. PhD thesis, University of Southern California, Department of Electrical Engineering Systems, USA, 1994.

[39] Taneda, A. Adplot: detection and visulization of repetitive patterns in complete genomes. *Bioinformatics 5* (2004), 701–708.

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