Paper category: Genetic Programming

What bloat? Cartesian Genetic Programming on Boolean problems

Julian Miller
School of Computer Science, University of
Birmingham, Birmingham, B15 2TT, UK
Telephone: +44 121 414 3710
Email: j.miller@cs.bham.ac.uk14 3710
Email: j.miller@cs.bham.ac.uk

Abstract

This paper presents an empirical study of the variation of program size over time, for a form of Genetic Programming called Cartesian Genetic Programming. Two main types of Cartesian genetic programming are examined: one uses a fully connected graph, with no redundant nodes, while the other allows partial connectedness and has redundant nodes. Studies are reported here for fitness based search and for a flat fitness landscape. The variation of program size with generation does not behave in a similar way to that reported in other studies on standard Genetic Programming. Depending on the form of Cartesian genetic programming, it is found that there is either very weak program bloat or *zero bloat*. It is argued that an important factor in the analysis of the change of program length is neutral drift, and that if genotype redundancy is present, the genetic neutral drift simultaneously improves search and compresses program code.

1 INTRODUCTION

In many reported forms of Genetic Programming (GP) special measures need to be taken to counteract the tendency of programs to rapidly grow in size over time. The latter is usually referred to as *bloat*. Uncontrolled bloat leads to time consuming fitness evaluation and also reduces the efficiency of the search operators. Consequently researchers have usually imposed some form of parsimony pressure to counteract this effect. This paper describes the variation of program size over time for a recently developed form of graph-based genetic programming called Cartesian Genetic Programming (CGP) [21][22].

The motivation for this paper grew out of the authors' observation that program bloat does not appear to occur in CGP. In fact it is found that programs only increase in size if it is necessary to improve their fitness. The paper sets out to try to understand this fact. A number of questions were immediately apparent. Does the imposition of a maximum program size in CGP provide a compression pressure that counteracts bloat? Is the form of mutation that is employed exerting a parsimony pressure?

At the outset the authors' intuition suggested that an important consideration in answering these question was the presence in CGP of genetic redundancy. In CGP there is a genotype-phenotype mapping stage and program nodes may be specified in the genotype that do not code for anything in the phenotype (the actual program). Moreover, these redundant nodes are not even evaluated during the assessesment of a program's fitness. The mutation operator can switch nodes into and out of the phenotype during the evolutionary process. It has been shown elsewhere that the presence of these redundant nodes considerably enhances the ability of the evolutionary algorithm to search effectively [22][30]. To investigate this intuition it was necessary to investigate the behaviour of a form of CGP in which redundant nodes were eliminated i.e fully-connected graphs. Attendant on this requirement was development of another mutation operator, in addition to the simple point mutation that would insert or delete program nodes. This in turn meant that both genotypes and phenotypes would have to have variable size. The simplest way to investigate whether point mutation was naturally parsimonious was to examine the case of a flat fitness landscape. Would the evolutionary algorithm be biased toward small programs on a flat landscape? It is these questions and intuitions that are investigated and explored in this paper.

The plan of the papers is as follows. Section 2 gives a brief survey of the program bloat phenomenon, and its relation to non-coding regions. Section 3 gives a description of CGP and its mutation operators. The experiments are described in section 4. Section 5 lays out the experimental findings. Some analysis and discussion of the results and how they relate to the behaviour of other GP paradigms is given in Section 6. In Section 7 conclusions are given. The paper concludes with a brief discussion of further work in section 8.

2 RELATED WORK

The problem of the rapid growth of programs produced by Genetic Programming is very well known and is generally referred to as program bloat [1][2][4][8][10][11][17][18] [27][28][29]. Unfortunately this growth in program size is almost always due to the growth of pieces of sub-code that

have little or no semantic effect. Various ideas have been proposed to explain this phenomenon. Originally it was viewed as hitchhiking [29] which viewed inactive code being propagated *by crossover,* by being attached to fitter parents. Another theory was that bloat arose because it provided a protection from the deleterious effects of crossover by increasing the number of crossover points that have no semantic effect on an individual [4][17][18][25]. Another argument put forward was that of *removal bias* [27][28]. This suggested that there was a natural bias toward large subtree growth because removal of the whole redundant subtree would be disruptive, while enlarging the inoperative inflated code would not change the fitness of the program. Many of the arguments have focussed on the crossover operator though there is no clear reason why these theories might not be similarly applied to mutation operators also. There has been work done that suggests that subtree crossover is particulary at fault and mutation to a smaller extent [2].

Much work has focused on the intron view of bloat. Introns are extraneous pieces of code that do not contribute to program fitness. One approach to alleviate the intron problem has been to deliberately insert introns, i.e. insert *explicitly* defined introns [16]. In register machine code GP this can have the effect of automatically supressing the growth of implicit introns. Recently however, work has been done that suggests that program growth is not caused by intron growth but rather intron growth is a *consequence* of program growth. The program growth is linked to the implicit bias in tree-based GP toward deep crossover points because disruption to subtrees near to the program root are likely to be deleterious [15]. This fits with the findings in [19] which showed that throughout the evolutionary run the nodes closest to the root hardly ever change from those in the initial population. Possibly the most general argument advanced is that "fitness causes bloat" which asserts that program bloat occurs largely because there are many more larger programs with higher fitness so the small initial programs drift in this direction [9][10][11][12][13]. This theory has matured to the point that now such growth is seen as an inevitable consequence of evolving variable length program representations for two main reasons: 1) search operators with no explicit length bias tend to sample bigger programs (see above), and 2) competition within populations favours longer programs as they can usually reproduce more accurately.

The presence of implicit introns in genetic programming is almost universally regarded as bad, yet some researchers have argued that the spread of introns can actually be beneficial in that they provide a natural kind of code compression [17]. It was partly to alleviate the drawbacks of implicit introns that they introduced explicitly defined introns. However recent applications of this idea to tree-based sytems has been less successful, and it seems that *supressing* certain types of implicit introns is more beneficial to search [26]. One early finding has shown that with genetic algorithms, non-coding regions actually improve the performance [14]. More recent studies have indicated that introns can improve performance but only with the imposition of a parsimony function [5].

3 CARTESIAN GENETIC PROGRAMMING

Cartesian Genetic Programming (CGP) was first formerly proposed in [22]. It shares some characteristics with Parallel Distributed GP (PDGP) [24]. It independently originated from work concerned with the design of digital circuits using evolutionary algorithms [20].

In CGP a program is represented as a rectangular array of nodes. The nodes represent any operation on the data seen at its inputs, and may implement any convenient programming construct (if, switch, OR, * etc.). All the inputs, whether primary data, node inputs, node outputs, or program outputs are sequentially indexed by integers. The functions of the nodes are also sequentially indexed. The chromosome is just a linear string of these integers. The idea is best explained with a simple symbolic regression example borrowed from [8]. Fig 1 shows the genotype and the corresponding phenotype for a program which implements both the difference in volume between two boxes V_1 - V_2, and the sum of the volumes, V_1 + V_2 , where, $V_1 = X_1X_2X_3$, $V_2 = Y_1Y_2Y_3$. The particular inputs corresponding to the dimensions of the two boxes X_1, X_2, X_3, Y_1, Y_2, Y_3, are labelled 0-5 and are seen on the left. The function set is {0=plus, 1=minus, 2=multiply}. The functions are shown in bold in the genotype and are seen inside the nodes. The program outputs are taken from node outputs 12 and 13. V_1 and V_2 are each re-used in the calculation of the two outputs. The inputs of columns of nodes can only be connected to the outputs (or program inputs) which are on the left. A node may have its imputs connected to the output of another node provided that the latter is no more than a certain number of columns back. This parameter is called *levels-back*. Using a levels-back =1 forces maximum re-use of individual node outputs but hampers large scale re-use of collections of nodes. On the other hand if levels-back = maximum number of columns (and there is only a single row) unrestricted connectivity of nodes is allowed. In this representation an output of a node may not be connected to the input of another node in the same column. Primary inputs (0-5) are allowed to connect to any node without restriction.

An important aspect of the representation is that some genes may not be expressed in the phenotype program. In CGP the distinction between non-coding genes and coding genes is purely in whether at that particular instance the node genes associated with the node's output are active because the node is connected between the program inputs and outputs. Genes can be made active by mutation at one point and then later made inactive. It is important to note that inactive genes are *not processed* when the fitness of a program is assessed. Of course introns can still occur in the phenotype but they do not appear to cause a problem.

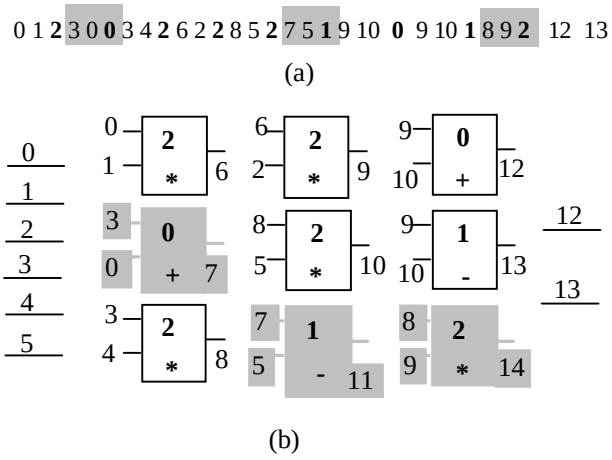


Figure 1: An example CGP genotype (a) and phenotype (b). Note that the nodes in grey do not form part of the phenotype.

In the example shown all the nodes have the same number of inputs; this is a convenience, not a fundamental requirement. Thus the representation could be readily generalised to accommodate variable number of inputs and outputs for each node. Since nodes do not have to be connected the number of nodes used can vary from 0 to the maximum number available. Thus bounded variable length programs are allowed. One of the other advantages of this representation of a program is that the chromosome representation used is independent of the data type used for the problem, as the chromosome consists of addresses where data is stored. Point mutation is very simple; one merely has to allow changes to the genes which respect either the functional constraints or the constraints imposed by levels-back. Note that output connections can also be mutated (e.g. 12 and 13 in Fig 1). In this study two other forms of mutation have been defined: *insert-node* and *delete-node*. Insert-node selects a position at random and inserts a new node with a randomly chosen function (from those allowed) and randomly chosen input connections. The delete-node operator randomly selects a node and removes it. After either operator is applied to the genotype the node output numbers are incremented or decremented to the right of the inserted or removed node so that the graph structure is disrupted as little as possible. This is illustrated in Fig 2.

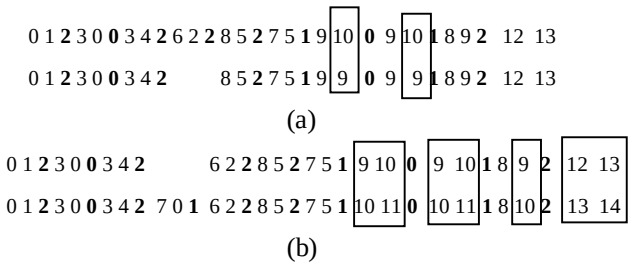


Figure 2: Genotype before and after node deletion (a), and before and after node insertion (b). Altered genes are indicated.

4 EXPERIMENTS

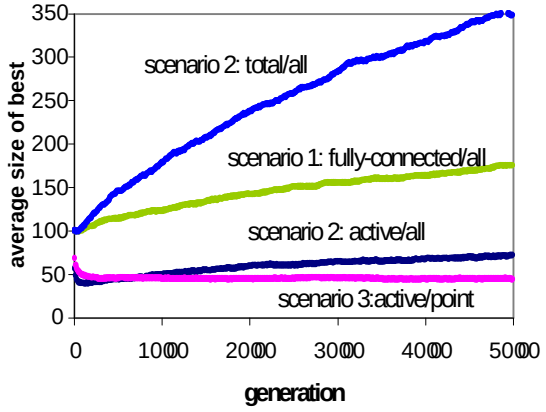
In this paper a special case of CGP appropriate for Boolean problems is employed where the data type is binary and the network is allowed to be feed-forward only. The problem chosen for this study is the three-bit multiplier which multiplies two three-bit binary numbers and outputs the corresponding six-bit binary number. The function set chosen for the experiments was {**AND, EXOR, IF, IF***}. All function nodes were assumed to have arity 3 (in the case of **AND** and **EXOR** the third input was ignored). The three input node **IF** with inputs A, B, C, implements the following function: **if** C **then** B, **else** A. The function IF* implements: **if** C **then NOT** B, **else** A. The three-bit multiplier problem was chosen for this study for two main reasons: it is a very challenging problem, and it requires a moderate number of nodes (at least 21 using the above function set).

All the experiments performed in this paper used a single row of nodes with levels-back set to the number of columns. Two forms of Cartesian programs were investigated. The first was the normal representation already described. The other was *fully-connected*. In this representation every node is connected to the node on the immediate left (excluding the leftmost node which can only connect to the program inputs). Thus at least one non-redundant node input was connected to its left neighbour. If this connection was mutated by point-mutation then *one* of the node inputs was randomly chosen to be the new left-neighbour connection and the other subjected to the point mutation. This was implemented so that connections would be free to move rather than remaining fixed throughout the evolutionary run. If the *insert-node* added a node *j* between successive nodes *i* and *k* then *j* had to have an input connected to the output of *i* and the input of node k that had formerly been connected to the output of node *i* would then connect to *j*. All other nodes would be connected as before the node insertion was carried out. Thus a fully-connected graph would be obtained with the most similarity to the original. If *delete-node* was applied (so that node *k* was removed) then the remaining graph (from node *j* rightwards) would move left and one of inputs of node *j* would be randomly chosen to connect to *i*. Clearly, if either insert/delete-node operators were applied then the number of columns would also be changed. Thus variable length structures were employed.

The evolutionary algorithm used was of (1+1) evolutionary strategy. In each iteration a genotype was randomly chosen with fitness *equal or greater* to the previous best (the new parent). This was then mutated to form the child. According to the mutation rate (2% was chosen for all experiments) a certain number of genes would be mutated, and only 50% of these genes underwent point-mutation. The remaining genes chosen for mutation then underwent insert/delete-node mutation with equal probability.

5 RESULTS

In the first series of experiments the population was initialised with fully connected programs of 100 nodes. Three scenarios were examined. Scenario 1 allowed only fully connected graphs with all three mutation types: point, insert-node, delete-node. This is referred to as fully-connected/all. Scenario 2 did not require that graphs after the initial population be fully-connected . All three mutation types were again used. The third scenario again did not require graphs to be fully connected after the initial population but only employed point mutation. One hundred runs were carried out in each scenario with 50,000 generations. It is important to note that the fully-connected scenario the size of the program (number of active nodes) is the same as the total number of nodes. In the second scenario the total number of nodes can change (due to the action of insert/delete-node mutation), however not all nodes are active (i.e involved in the phenotype). In the last scenario however the total number of nodes is fixed at 100 but the number of active nodes can vary between 0 and the maximum value (100). Graphs showing the variation of program size with generation for various Boolean functions



are shown in Figs 3, 4 and 5.

Figure 3: Variation of program size with generation for three-bit multiplier under various mutation operators.

Observing the behaviour in scenario 1, it is seen that there is a fairly slow bloat. It is much less than the near quadratic bloat familiar in certain forms of standard tree-based GP [13]. In scenario 2 there is a rapid decrease in phenotype size which is followed by a slow increase. However the genotype increases in size much more rapidly and without an intial decrease. In scenario 3, again there is a rapid decrease in phenotype size to an almost constant value. In scenario 2 there is a higher probability that a node deletion will result in poorer fitness than a node insertion. If a mutation results in an active node being deleted it will be very disruptive to the phenotype. However inserting a node may have no effect as it may not become active. Since genotypes with more inactive nodes have the same fitness there is a chance that it will be chosen to replace the parent genotype.

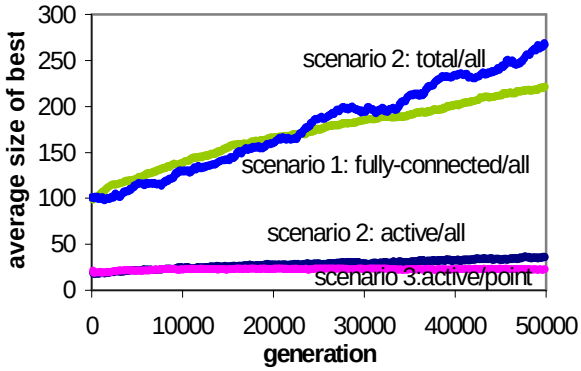


Figure 4: Variation of program size with generation for even-five parity under various mutation operators.

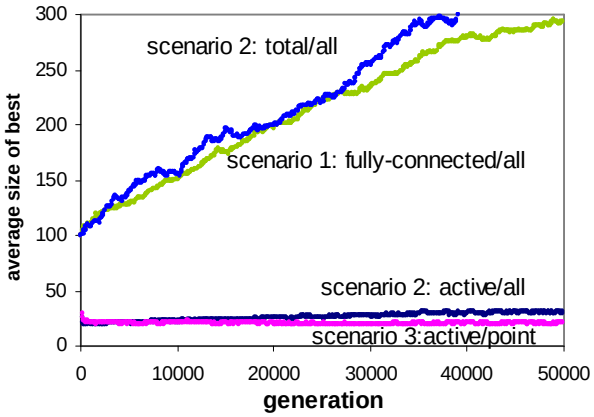


Figure 5: Variation of program size with generation for 6-mux under various mutation operators.

This may explain the genotype growth in scenario 2. In other studies using CGP [30] [31] it has been shown that by assisting neutral drift, genotype redundancy allows a greater exploration of phenotype space and hence leads to higher fitness. In scenario 2 the algorithm can increase the redundancy by increasing the total length of the genotype which increases the opportunity for neutral drift. In scenario 1, neutral drift is still possible but only by increasing the size of non-coding sections of phenotype (rather like *implicit* bloat in conventional tree-based GP). In scenario 3, where only the point mutation is operative, average program size remained almost constant with time. In scenario 2 data was collected on the behaviour of the largest and smallest eventual genotypes and their corresponding phenotype lengths. This is shown in Figs 6 and 7. Note, data was collected at each fitness improvement.

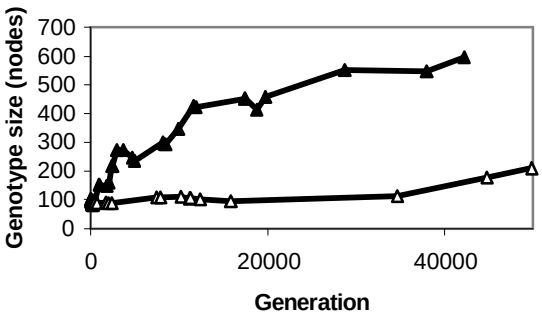


Figure 6: Variation of genotype size with generation for two extremal runs under point, insert-node, and delete-node mutation operators (scenario 2)

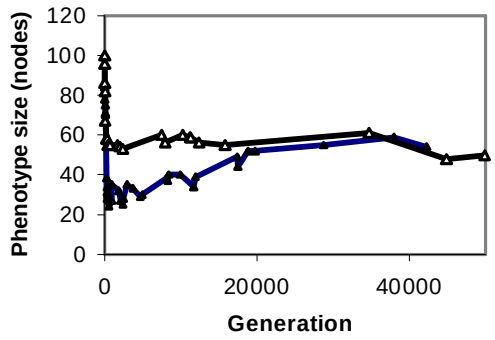


Figure 7: Variation of phenotype size with generation for two extremal runs under point, insert-node, and delete-node mutation operators (scenario 2)

Figures 6 and 7 are quite revealing. In the run that eventually produced the largest genotype (black triangles) the phenotype decreases much more rapidly from its initial size (100 nodes) than the other run. Consequently it became even more probable that the node insertion operator would be less disruptive than the deletion operator and hence more redundant nodes would be inserted into the genotype and it would begin to bloat. In the experiments described above the average fitness of the best in population was measured. The results are shown in Figure 8.

The average fitness of the initial population is, of course, the same in all scenarios (241.51, not shown). After 20000 generations differences between the three scenarios become statistically significant. Scenario 3 gives the highest average fitness, scenario 2 is next and scenario 1 is worst. Clearly evolving fully-connected graphs is considerably less effective than allowing genotypes that include redundant nodes. Why should this be? It has been already pointed out that the proportion of programs with a given fitness is approximately constant for a wide range of program lengths[9][10][11][12][13]. Since the total number of programs rises rapidly with length, the number of programs with a given fitness must also increase rapidly. Thus one should expect larger programs to be favoured. If this argument applies equally to graphs in CGP then one would expect the size of all the graphs in the experiments described above to increase rapidly, and possibly with a near quadratic dependence on generation [13]. This clearly does not happen. The fact that in scenario 3 the programs all have an upper bound (100 nodes) does not appear to supress program growth after the initial rapid drop (Figs 3 and 4). There is still plenty of room to grow from about 40 nodes, yet program size remains approximately constant with time.

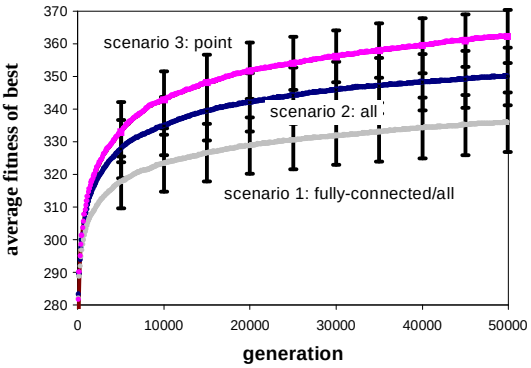


Figure 8: Variation of fitness with generation for three-bit multiplier under various mutation operators (error bars are ± standard deviation)

In molecular evolution in biology Kimura [7] has observed that nearly all mutations result in genotypes with the same fitness, and that genetic drift is a large causative factor for large phenotypic diversity. The experimental results shown here support the idea that neutrality is also a factor in the tendency for program length to increase with evolutionary step. It may be that the genetic drift that leads to fitness improvement is so important that in Cartesian GP *external* bloat occurs in the redundant code thus automatically compressing the phenotypic code (scenarios 2 and 3 above). It might however, be argued that perhaps the point mutation operator introduced a parsimony pressure that counteracted the natural tendency of programs to bloat. In order to test this and further shed light on the above arguments, the experiments were run again, but this time with fitness switched off (the fitness function returned 1 for all genotypes). The results are shown in Fig 9.

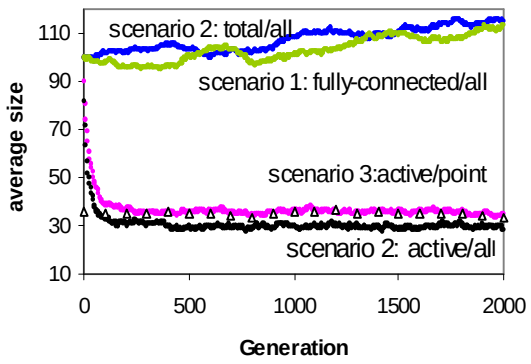


Figure 9: Variation of program size with generation under various mutation operators for flat fitness landscape

Firstly in the experiments that involve variable length genotypes (scenarios 2 and 3) the size of the genotype varies around the initial figure, though there is a slight tendency for the size to very slowly increase. The cause of this isn't clear, it may be analogous to Brownian motion. It could also be due to the slight non-randomness in the pseudo-random number generator used (evolution with flat fitness landscape is *very* sensitive to this). In scenarios 2 and 3 once again a rapid contraction of phenotype size occurs. Subsequently the size tends to a constant value. In the case of scenario 3 this value is approximately 35 nodes. In scenario 2 this value is approximately 30 nodes. Immediately below the plot of active size in scenario 3 is seen the plot of average active size under point mutation (triangular symbols) when the populations are not initialised to programs with 100 fully-connected nodes (instead 100 nodes is the maximum allowable size). Clearly even when initialised to fully-connected nodes the point mutation operator quickly reduces the average size of the phenotypes to that of randomly initialised populations. Since the average size remains approximately constant it can be inferred that point mutation is an unbiased operator and does not exert any parsimony pressure on randomly initialised populations.

The minimum number of nodes required to build the three-bit multiplier using the function set {**`AND, EXOR, IF, IF*}`**appears to be 21 [23]. Thus in all the experiments so far described the allowed number of nodes was greatly more than is required. It was interesting to examine how the program size might evolve under point mutation if initially the average size was less than 21. Larger programs would then be favoured by the algorithm as they would be the correlated with an increase in fitness.

Two further experiments were carried out (each 100 runs and 4% mutation) to examine the average size of the best in population. A higher mutation rate was chosen because 2% would have only allowed one gene in each population member to be mutated per generation. In one set of runs the initial population consisted of 30 fully-connected nodes. In the other the population was initialised randomly with a maximum size of 30 nodes. This gave an average phenotype size of 15.21 nodes. The results are plotted in Fig 10.

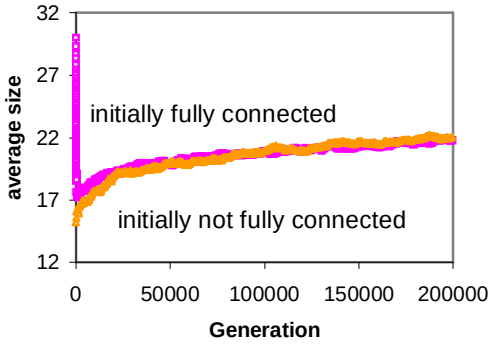


Figure 10: Variation of program size with generation under point mutation and different initialisation conditions with a maximum of 30 nodes

Once again a rapid decrease in average size is observed when the programs are initialised to be 30 nodes. Its smallest average size however remains a little larger than when the population is randomly initialised. In both scenarios there is a slow increase in average program length that appears to level off at about 22 nodes.

6 DISCUSSION

The view that sees neutral drift as a causative factor in program bloat has received little attention in the literature. Programs that have varying amounts of junk code within them all have the same fitness. Evolutionary algorithms, unlike strict hillclimbers (which don't exhibit bloat [13]), do not typically demand that promotable programs (to the next generation) have an improved fitness, thus they may accept equally good solutions (i.e. fitness neutral) or even slightly worse solutions. Consequently, if there is a mechanism that can create neutral solutions a genetic drift process will occur, particularly during periods of no fitness improvement (which is when *implicit* bloat can be at its worse [16]). In program representations that do not distinguish genotype from phenotype (i.e standard tree-based GP) this process of drift must largely occur by the insertion of junk code. In other work [30][31] it has been shown that genetic drift is highly beneficial in CGP as it allows constant innovation and removes genetic stagnation. This is also observed in other systems [3][6]. However genetic drift with implicit introns appears to cause stagnation and supresses constant innovation. One advantage of making a distinction between genotype and phenotype is that the exploratory nature of genetic drift can occur mainly in fitness neutral space and only occasionally affect phenotype space. This means that there is no penalty associated with the neutral exploration as

it is never evaluated when the fitness of a program is calculated. The argument that program bloat provides a protective mechanism for the destructive effects of both crossover and mutation (i.e. it is a *good* thing) applies equally well to explicit redundancy. Thus one can take advantage of it in CGP without paying the penalty of evaluating it. To some extent one can see fully-connectedness as an invitation for program bloat and it is really difficult to see any virtues it may have over representations that allow explicit code redundancy.

Standard CGP (without insert/delete-node operators) has a bounded program size. However this does not seem to be a large factor in program size suppression as in a flat fitness landscape the average size of the programs is always a fraction of the maximum bound (see Figs 9 and 10). Clearly it would be a problem in a fitness based search if the bound chosen was less than the minimum size to construct a correct program. A suggested remedy for this is given in the further work section.

7 CONCLUSIONS

This paper has briefly surveyed the published literature on the evolution of program size and contrasted the reported behaviour with that of new form of genetic programming called Cartesian Genetic Programming. Experiments performed indicate that implicit intron growth is not a problem and no measures need to be taken to suppress it (at least for some Boolean problems). Evidence has been provided of the unbiased nature of the mutation operator by examining the behaviour of the programs under evolution in a flat fitness landscape. The central concept of the work is that allowing unconnected program nodes is very useful and improves the effectiveness of the search without having to be evaluated in the fitness function. Such representations *benefit* from explicit introns which allow program exploration through genetic drift.

8 FURTHER WORK

The question as to which kinds of explicit introns are best and why, and their role in suppressing bloat and allowing innovation needs more detailed investigation. Variable length program representations have an obvious advantage over bounded length representations (standard CGP) in that one can start the evolution with relatively small programs that consume little memory and are quick to process. However the work of this paper definitely implies that allowing mutation operators to be the mechanism for this length variability is liable to produce poorer fitness improvement and extra processing. Instead one could introduce length variability by inspecting the phenotypic length and, if it became "too close" to genotypic length, then increase genotypic length by randomly introducing program nodes. The investigation of this remains for the future.

Acknowledgments

Many thanks to Tina Yu, Jon Rowe and especially Nick McPhee for their comments and suggestions for improvement.

References

1. L. Altenberg (1994). Emergent phenomena in genetic programming. In A. V. Sebald and L. J. Fogel (eds.), *Evolutionary Programming: Proceedings of the Third Annual Conference*, 233-241. World Scientific Publishing.
2. P. J. Angeline (1998). Subtree crossover causes bloat. In J. R. Koza et al (eds.), *Genetic Programming 1998: Proceedings of the Third Annual Conference*, 745-752. Morgan Kaufmann.
3. L. Barnett (1998). Ruggedness and Neutrality - the NKp family of Fitness Landscapes. In: C. Adami, R. Belew, H. Kitano, and C. Taylor (eds.) *Proceedings of the Sixth International Conference on Artificial Life*, 18-27. MIT Press.
4. T. Blickle and L. Thiele (1994). Genetic programming and redundancy. In J. Hopf (ed.), *Genetic Algorithms within the Framework of Evolutionary Computation* (Workshop at KI-94), Saarbrücken), 33-38. Max-Planck-Institut für Informatik (MPI-I-94-241).
5. D. S. Burke, K. A. De Jong, J. J. Grefenstette, C. L. Ramsey, and A. S. Wu (1998). Putting More Genetics into Genetic Algorithms. Evolutionary Computation, Vol. 6, No. 4, 387-410.
6. M. A. Huynen (1996). Exploring Phenotype Space through Neutral Evolution. *Journal of Molecular Evolution* Vol. 43, 165-169.
7. M. Kimura (1968). Evolutionary Rate at the Molecular Level. *Nature* Vol. 217. 624-626.
8. J. R. Koza (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MIT Press.
9. W. B. Langdon (1998). The evolution of size in variable length representations. In *1998 IEEE International Conference on Evolutionary Computation*, 633-638. IEEE Press.
10. W. B. Langdon, and R. Poli (1998). Fitness Causes Bloat: Mutation. In W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty (eds.), *EuroGP'98: First European workshop on Genetic Programming*, 37-48. Springer-Verlag.
11. W. B. Langdon, and R. Poli (1998). Why Ants are Hard. In J. R. Koza et al (eds.), *GP'98: Proceedings of the Third Annual Genetic Programming Conference*, 193-201. Morgan Kaufmann.
12. W. B. Langdon, T. Soule, R. Poli, and J. Foster (1999). The Evolution of Size and Shape. In L Spector, W. B. Langdon, U-M. O'Reilly, P. J. Angeline (eds.),

Advances in Genetic Programming Vol. 3, 163-190. MIT Press.

13. W. B. Langdon (2000). Quadratic Bloat in Genetic Programming. In D. Whitley, D. Goldberg, E. Cantú-Paz, L. Spector, I. Parmee, and H-G Beyer, *GECCO-2000: Proceedings of the Genetic and Evolutionary Computation Conference*, 451-458. Morgan Kaufmann.
14. J. R. Levenick (1991). Inserting introns improves genetic algorithm success rate: Taking a cue from biology. In R. K. Belew and L. B. Booker (eds.), Proceedings of the Fourth International Conference on Genetic Algorithms, 123-127. Morgan Kaufmann.
15. S. Luke (2000). Code growth is Not Caused by Introns. In *GECCO-2000: Late Breaking Papers*, 228-235. Morgan Kaufmann.
16. P. Nordin, F. Francone, and W. Banzhaf (1995). Explicity defined introns and destructive crossover in genetic programming. In Peter J. Angeline and K. E. Kinnear Jr. (eds.), *Advances in Genetic Programming* Vol. 2, 111-134. MIT Press.
17. P. Nordin, and W. Banzhaf (1995). Complexity compression and evolution. In L. Eshelman (ed.), *Genetic Algorithms: Proceedings of the Sixth International Conference(ICGA95)*, 310-317. Morgan Kaufmann.
18. N. F. McPhee and J. D. Miller (1995). Accurate replication in genetic programming. In L. Eshelman (ed.), *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, 303-309. Morgan Kaufmann.
19. N. F. McPhee and N. J. Hopper (1999) Analysis of genetic diversity through population history, In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith: *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO99)*, 1112-1120. Morgan Kaufmann.
20. J. F. Miller, P. Thomson, and T. C. Fogarty (1997). Designing Electronic Circuits Using Evolutionary Algorithms. Arithmetic Circuits: A Case Study. In D. Quagliarella, J. Periaux, C. Poloni and G. Winter (eds.), *Genetic Algorithms and Evolution Strategies in Engineering and Computer Science: Recent Advancements and Industrial Applications*, chapter 6. Wiley.
21. J. F. Miller (1999). An empirical study of the efficiency of learning Boolean functions using a Cartesian Genetic Programming Approach, In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith (eds.), *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO99)*, 1135-1142. Morgan Kaufmann
22. J. F. Miller and P. Thomson (2000). Cartesian Genetic Programming. In R. Poli, W. Banzhaf, W. B. Langdon, J. F. Miller, P. Nordin, T. C. Fogarty, (eds.),*Third European Conference on Genetic Programming. Lecture Notes in Computer Science*, Vol. 1802, 121-132
23. J. F. Miller, D. Job, and V. K. Vassilev (2000). Principles in the Evolutionary Design of Digital Circuits -- Part I. *Journal of Genetic Programming and Evolvable Machines, Vol. 1*, No. 1, 8-35. Kluwer Academic.
24. R. Poli, (1997). Evolution of graph-like programs with parallel distributed genetic programming. In T. Bäck (ed.), *Genetic Algorithms: Proceedings of the Seventh International Conference (ICGA96)*, 346-353. Morgan Kaufmann.
25. J. Rosca (1996). Generality versus size in genetic programming. In J. R. Koza et al (eds.), *Genetic Programming 1996: Proceedings of the First Annual Conference*, 381-387. MIT Press.
26. P. W. Smith and K. Harries (1998). Code growth, explicitly defined introns, and alternative selection schemes. *Evolutionary Computation*, Vol. 6, No. 4, 339-360.
27. T. Soule, J. A. Foster, and J. Dickinson (1996). Code growth in genetic programming. In J. R. Koza et al (eds.*), Genetic Programming 1996: Proceedings of the First Annual Conference*, 215-223. MIT Press.
28. T. Soule (1998). *Code Growth in Genetic Programming*. PhD thesis, University of Idaho.
29. W. A. Tackett (1994). *Recombination, Selection, and the Genetic Construction of Computer Programs*. PhD thesis, University of Southern California.
30. V. K. Vassilev, and J. F. Miller (2000). The Advantages of Landscape Neutrality in Digital Circuit Evolution. In J. F. Miller, A. Thompson, P. Thomson, and T. C. Fogarty T. C. (eds.), *Proceedings of the Third International Conference on Evolvable Systems: From Biology to Hardware (ICES2000)*, Lecture Notes in Computer Science, Vol. 1801, 252-263. Springer
31. T. Yu, and J. F. Miller (2000). Neutrality and Evolvability of a Boolean Function Landscape. In J. F. Miller, M. Tomassini, W. Langdon (eds.), *EuroGP'2000: Fourth European Conference on Genetic Programming*. Lecture Notes in Computer Science, Vol. 2038, 204-217. Springer-Verlag.