Genetic Programming: A Review of Some Concerns

Maumita Bhattacharya and Baikunth Nath

School of Computing & Information Technology
Monash University (Gippsland Campus), Churchill 3842, Australia
{Email: Maumita,Baikunth.Nath@infotech.monash.edu.au}

Abstract: Genetic Programming (GP) is gradually being accepted as a promising variant of Genetic Algorithm (GA) that evolves dynamic hierarchical structures, often described as programs. In other words GP seemingly holds the key to attain the goal of "automated program generation". However one of the serious problems of GP lies in the "code growth" or "size problem" that occurs as the structures evolve, leading to excessive pressure on system resources and unsatisfying convergence. Several researchers have addressed the problem. However, absence of a general framework and physical constraints, viz, infinitely large resource requirements have made it difficult to find any generic explanation and hence solution to the problem. This paper surveys the major research works in this direction from a critical angle. Overview of a few other major GP concerns is covered in brief. We conclude with a general discussion on "code growth" and other critical aspects of GP techniques, while attempting to highlight on future research directions to tackle such problems.

1. Introduction

One of the major drawbacks of Genetic Programming (GP) coming in its way of becoming the dream "machine" for automated program generation is "Code Growth". If unchecked this results in consumption of excessive machine resources and slower convergence, thus making it practically unfit for complex problem domains needing large individuals as member programs in the search space. A good deal of research has gone into identifying causes and solutions to this problem [3,8,9,12,14,18]. However, as real life applications demand, absence of commonality in complexity of problem domain, operator structures, basic representations used in these works and the ultimate bounds on feasible population size, have failed to produce any clear-cut explanation of this problem. This paper analyses the "code growth" and a few other major GP concerns, research inputs in this direction and their shortcomings. An attempt has been made to highlight future research directions in this regard.

The discussion starts with a brief analysis of a few major genetic programming concerns relating to contribution and behavioral characteristics of important GP operators like crossover and mutation, and the issue of generality of genetic programming solutions (section 2 and 3). Detailed discussion on basic GP techniques and related operators can be found in [3].

The subsequent sections are devoted to a brief discussion on the "code growth" problem in general, while critically analyzing some of the research inputs in this

V.N. Alexandrov et al. (Eds.): ICCS 2001, LNCS 2074, pp. 1031–1040, 2001.

direction. A brief summary of other GP constraints and probable future research directions are covered in section 5 and 6.

2. The Controversy over GP Operators

The major issues related to the breeding operators are concerned with contributions and behavioral characteristics of these operators. The following discussion deals with a comparative analysis of the crossover and the mutation operator.

The Tussle Between Mutation and Crossover: In GP, since variable-length representation is involved, choosing a better operator means, picking the one, yielding the highest overall fitness before bloat stops all effective evolutions. While fitness and mean tree size are both highly domain-dependent factors, researches [17] have shown that a tenuous trend does exist [figure 1].

Inference 1: *In general, population size more strongly determines whether crossover produces better fitness, whereas, number of generations more strongly determines whether it produces larger trees.*

Inference 2: *Crossover is apparently a more efficient approach for large populations with small run lengths. Mutation on the other hand, is more efficient for small population with long run lengths.*

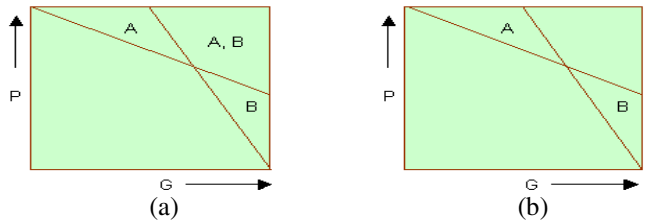

Fig. 1. (a) Areas where Subtree Crossover has better fitness, or where Subtree Mutation has better tree size. Here, A: Area where Crossover has higher fitness and B: Area where Mutation has smaller tree size. (b) Areas where Subtree Crossover or Subtree Mutation is more efficient. Here, A: Area where Crossover is more efficient and B: Area where Subtree Mutation is more efficient. P and G denote the population size and the number of generations respectively.

Whither Crossover: Though an important breeding operator, crossover may not be the best choice in every situation. One of the "inadequacies" of crossover is realized in the form of loss of diversity in the population, and thereby forcing GP to rely on the ineffective mutation operator to create "fitter" solutions [2]. Let us consider GPs with function sets of function arity two. Let us consider the following:

d - the maximum tree depth; l - tree layer from root node to d ;

n_l - the number of nodes in layer $l = 2^l$, where $0 \le l \le d$;

N_l - the number of nodes in all layers from root to layer l = $\sum_{i=0}^{l} n_l = \sum_{i=0}^{l} 2^i = 2^{l+1} - 1$;

N_d - the total number of nodes in a tree of depth $d = 2^{d+1} - 1$;

$\mathrm{P(any)}_1$ - the likelihood of any node(s) in a tree, in layer l, experiencing some crossover activity, i.e. when crossover point occurs in any layer from 0 to l = $\frac{N_l}{N_d} = \frac{2^{l+1}-1}{2^{d+1}-1} \cong \frac{1}{2^{d-l}}$;

$\mathrm{P(layer)}_1$ - the likelihood of crossover occurring within a layer l, i.e. when the two crossover points are in the same layer

$= \frac{{n_l}^2}{{N_d}^2} = \frac{2^{2l}}{(2^{d+1}-1)^2} \cong \frac{2^{2l}}{2^{2d+2}} = \frac{1}{2^{2(d-l+1)}}$

$\mathrm{P(2\ legal\ offsprings)}$ - the likelihood of crossover based on a random choice of crossover points, producing two legal offspring

$= \frac{legal}{total} = \frac{\sum_{i=0}^{d} {n_i}^2}{{N_d}^2} \cong \frac{\sum_{i=0}^{d} 2^{2i}}{2^{2(d+1)}} \cong \frac{1}{4}$, for large d;

From the above calculations, analyzing $\mathrm{P}(any)_l$, we find that the upper layers receive relatively little attention from the crossover operator, especially as the tree grows large. As subtree discovery and the spread of subtrees takes place at lower levels, chiefly involving the leaf nodes, immediately beneficial subtrees are quickly spread within the trees at the expense of other subtrees of less immediate benefit. This could lead to either faster convergence or loss of diversity in the population.

3. The Generality Issue

Generality of solutions can be treated as one of the major guiding principles for the search of representations using genetic programming. Here, generality can be expressed as how well a solution performs on unseen cases from the underlying distribution of examples. Generalization may be easier to achieve with small solutions, but it is hard to achieve, just by adding size component in the fitness function. Biasing search towards smaller programs, in the pursuit of finding compact solutions, also leads to biasing towards a model of lower dimensionality. A biased model has a fewer degrees of freedom and lower variance. We normally aim to achieve a low bias and low variance model, but in practical situations, there has to be a trade off between the two.

For standard GP, the structural complexity is simply the number of nodes in the tree representation. For GP, using subroutines, it is computed by adding up the structural complexities of all subroutines or ADFs. The average path length of an individual i can be defined as $\overline{l(i)} = \frac{1}{n(i)} \sum_{j=1}^{n(i)} dist(root(i), j)$,

Where, $n(i)$ is the number of leaves of the tree representing i and $dist(root(i), j)$ is the distance between two nodes. Researches [4] findings are as below:

• *The average path length of an offspring is crossover independent on an average, if only a single offspring is generated from each selected couple.*

• *Mutation, on the other hand is path length invariant on average over generations, only if newly generated subtrees have the same size as replaced subtrees.*

These size-operation links have enormous effect on the generality issue. Effective size offers an insight into the properties of the search space in a given problem domain. More importantly, this can be used for modularization purposes leading to a smaller complexity and increased generalization.

4. The "Code Growth" Problem

The problem of "code growth" arises, as the evolved structures appear to drift towards large and slow forms on an average. It is a serious problem as it causes enormous pressure on resources (in terms of storage and processing of extraneous codes) that is disproportional to the performance of the evolving programs. It even may reach a point where extraneous codes may invade the search space to such an extent that improving useful codes will become virtually impossible. This problem has been identified under various names, such as "bloat", "fluff" and "complexity drift" or increasing "structural complexity"[19]. The questions, which demand answers, are:

- *Cause of code growth - Is it a protective mechanism against destructive Crossover?*
- *Role played by "complexity drift" in selection & survival of structures.*
- *Role of other operators in producing code growth.*
- *Relationship between code growth and code shape.*
- *Relationship between code growth and code shape.*
- *Affectivity of parsimony pressure in controlling code growth.*

Most of the above mentioned concerns have been dealt with by Koza [3], Altenberg [6], Sean Luke[17], Blickle and Thiele [12], Nordin and Banzhaf [9], Nordin [11], McPhee and Miller [7], Soule et al. [14], Langdon et al. [18]. The common approach to address the problem has been either by enforcing a size or depth limit on programs or by an explicit size component in the GP fitness [3, 1, 5]. Alternate techniques have been proposed by Blickle [16], Nordin et al. [10]], Soule and Foster [14], Soule [15],

Angeline [8], Langdon [20]. However, as mentioned earlier, absence of a general framework makes it difficult to draw a generic conclusion about the "code growth" problem.

The Problem

Various research works have shown that program depth grows apparently linearly but rapidly with standard crossover. While being problem and operator dependent, on average program trees grow roughly by one level per generation, ultimately putting tremendous pressure on system resources [20]. It has been observed that if the program size exceeds some problem and fitness level dependent threshold limit, the distribution of their fitness value in the GP search space does not get affected by "complexity" or length. Hence the number of programs with a specific fitness is distributed like the total number of programs and most programs will have a depth close to $2\sqrt{\pi(\text{number of internal nodes})}$, ignoring terms $O(N^{1/4})$. Any stochastic search technique, inclusive of GP, finds it difficult to improve on the best trial solution and the subsequent runs are likely to produce either same or even worse trial solutions. The selection process ensures that only the better ones survive. Unless a bias is used, the longer programs with the current level of performance will have better chance to survive.

What Is Behind It

As has been mentioned earlier, several researchers have tried to find an answer to this question. Based on the chosen problem domain, types of operators various inferences have been drawn in this regard. While considering "code growth", its worthwhile to consider both the average program size and the size of the largest program. Both of these increase considerably, in a typical GP program [15, 20, 11]. In standard GP, only the crossover and the selection operators affect the population. A crossover operation affects only the individual programs within a population, but apparently does not have any direct effect on the average size of the population. On the other hand, the selection operator can easily change the average size of the population by preferentially selecting the larger or the smaller programs. Hence, the selection operator can only increase the average program size to the largest program size, and then by selecting several copies of the largest program. Thus, apart from other factors, the sustained "code growth" in GP can only possibly be explained by combined effect of these two operators or an interaction between the two [15]. Effect of inclusion of fitness over code growth has been discussed by Langdon and Poli [19]. According to them, random selection did not cause bloating. However, "code growth" does not directly contribute to fitness, as most of the excess codes are nothing but redundant data. Presently there are several distinct theories explaining the cause of "code growth", of which the major ones are [15, 10, 18]:

- *The Destructive Crossover Hypotheses*
- *The solution distribution theory*
- *Removal of bias theory*
- *The shape-size connection*

a) The Destructive Crossover Hypothesis

According to this hypothesis, on average, crossover has neutral or destructive effect on individuals as part of each parent gets discarded.

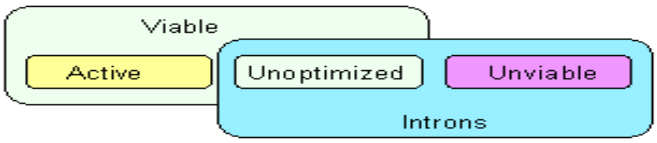

Fig. 2. A Venn Diagram Depicting Various Types of Codes and Their Relations

However, programs having a large amount of redundant data, stands less chance of getting discarded, as crossover has less effect on unviable codes [figure 2] and hence ensures better chance of survival for the individual [15, 14,17]. The probability p_s, of an individual being selected and surviving crossover is:

$$p_s = \frac{f_i}{\overline{f}} (1 - p_c p_d)$$

Where, p_c and p_d are the crossover probability and the probability that crossover is destructive. Let f_i and $\overline{f}$ be the i[th] individual's fitness and the average fitness of the population respectively.

b) The Solution Distribution Theory

This theory claims that "code growth" is partially caused by the distribution of semantically equivalent solutions in the phenotype or the solution space. In case of variable length representations, various semantically similar, but syntactically different individuals can represent a particular solution. A preferential choice towards larger solution will automatically lead to "code growth".

c) The Removal of Bias Theory

Subtree removal is the first step in both crossover and subtree mutation. This theory is based on the fact that changes in unviable code do not affect fitness and hence the source and size of a replacement branch during crossover, will not affect the fitness of the new program. Hence, there is no corresponding bias towards smaller (or larger) replacement branches. Fitness neutral operations will favor the removal of small branches and replacement with averaged sized branches creating a general increase in fitness neutral offspring. Besides, since crossover is either neutral or destructive in nature, these fitness-neutral offspring will be favored at the time of selection, thereby increasing the average program size, even in the absence of increase in fitness. Figure 3 depicts a generalized GP syntax tree [15]. The ancestors of any viable node are viable and if any node in a tree is viable, then the root node is definitely viable.

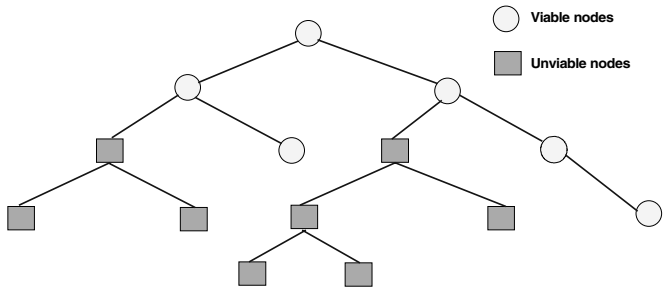

Fig. 3. A Generalized GP Syntax Tree

d) The Shape-Size Connection

Apart from the above three causes of "code growth" a shape-size connection may also be a contributor to "complexity drift". Code growth leads towards a specific problem independent syntax tree shape. Syntax trees have been found to evolve into problem dependent shapes in absence of "bloating".

Average growth in program depth when using standard subtree crossover with standard random initial trees is near linear [20]. The rate is about 1 level per generation but varies between problems. When combined with the known distribution of number of programs, this yields a prediction of sub-quadratic, rising to quadratic, growth in program size. There is considerable variation between runs but on the average we observe sub-quadratic growth in size. This in a continuous domain problem, rises apparently towards a quadratic power law limit.

Discrete mean length $\leq$ O (generations^2)

Continuous $\lim_{g \to \alpha}$ mean length= O (generations^2)

However in large discrete programs, the whole population has the same fitness, even though its variety is 100%.

Most GP systems store each program separately and memory usage grows linearly with program size. i.e. in the range of O(generations$^{1.2}$) to O(generations^2). Run time is typically dominated by program execution time.

An Analysis of Some Major Research Inputs to "Code Growth" Problem

As has been mentioned earlier, the problem of "complexity drift" or "bloating" has been reported and analyzed by many researchers. The principal explanation for "bloat" has been the growth of "introns" or "redundancy". Bickle and Thiele[12, 16], Nordin et al.[9, 10], and McPhee et al.[7] have argued in favor of "introns", supporting its code protection property at times of "destructive" crossover. Major percentage of "code growth" seems to occur to protect existing, successful programs against the damaging effect of "destructive crossover"[15]. However, this code protection property does not guarantee selection of better training sets. Poli and

Langdon argue that the fraction of genetic material changed by crossover is smaller in longer programs [19]. Such local changes may lead to GP population becoming trapped at local peaks in the fitness landscape. An attempt has been made to generalize the problem of "bloating" by expanding it to cover any progressive search technique, using "discrete variable length" representation. In general variable length representations, allow plentiful long representations (in absence of parsimony bias) than short ones. They have further proceeded with a typical GP demonstration problem to show that it suffers from "bloat" with fitness-based selection, even when mutation is used in place of crossover. Again, this demonstration problem has been used to establish that "bloat" does not occur in the absence of "fitness-based" selection. However, generalization of this concept is perhaps questionable, as even fitness-free neutral selection might tend to favor longer programs and crossover operator may preferentially pick up smaller branches for replacement by averaged size branches. These two functions may have combined contribution to "code growth". Terence Soule [15] has suggested a size and shape relationship for "code growth". It is possible that sparser the shape of the syntax tree, higher the chances of upward drift in complexity. Whatever may be the cause of "code growth", it is obvious from the vast research inputs in this direction that standard GP technique does not essentially try to evolve "better" programs, rather the inclination being towards evolving programs those survive, regardless of their performances. Hence, having an explicit fitness function serves little purpose. Rather, the focus should be on basing reproduction on program behavior and assessing it dynamically. It is also important to explore the yet unknown forces that affect the complex, dynamic system of an evolving population.

Means to Control "Bloat"

Of the major mechanisms suggested for controlling "bloat"[19], the first and the most widely used one uses a universal upper bound either on tree depth [3] or program length. The second method exerts parsimony pressure (as discussed in Section 3) to control "code growth"[3, 1]. The third one involves manipulation of the genetic operations [19, 12, 16].

5. More Concerns for GP

The problem of "code growth" is not really any isolated problem for genetic programming. Effective identification of "redundancy", quantification of the destructive power of crossover operator, the "representation" problem, generalization of GP process to handle discrete, complex problem domains, affectivity of "building block" hypotheses and standard genetic operators are a few among a host of several other GP concerns. Besides, as a direct effect of measures taken to control "code growth", in the form of imposition of restriction on tree depth, may have adverse effect on GP performance. This could manifest itself by decreasing diversity in the population and thereby leading to a premature convergence to a sub-optimal solution. Diversity may tend to drop near the root node and thereby causing inability for GP to create "fitter" solution trees.

6. Conclusion

GP, as a stochastic process for automatic program generation has gained some early success. In many areas, including robot and satellite guidance, data mining, neural net design, circuit design, and economic strategy development, GP has produced satisfactory results. However, it has not yet scaled well to larger and more complex problems. "Code growth" has a major role to play in this. Our purpose has been to identify the typical characteristics of a few GP concerns and the "code growth" problem in particular, and explore future research direction to solve the problems.

It has been observed that due to enormous resource requirement, simpler problems needing small trees have been chosen to accommodate adequate number of nodes in search space. Hence, the problem remains un-addressed for complex, discrete domains requiring large candidate trees. Use of evolutionary techniques to decompose problems and evolve smaller trees to generate larger trees may be used. However, time and resource requirements will still pose a problem. Hybrid search techniques may be incorporated to hasten up search process. It has been further observed that use of different operators produces different "bloat" characteristics making it harder to identify general "bloating" trends. Absence of selection operator altogether, prevents bloating. Generation and use of new mutation and crossover operators or even altogether new operators may help reducing "bloating". Inter-generational processing on fluctuating numbers of individuals may be used to reduce redundancy. Representation is another critical issue in genetic programming. In case of "directed acyclic graph" representation, memory usage may be less, as in every generation individuals are deleted. Re-evaluation of unchanged codes may be avoided by caching relevant information using conventional storage structure and fast sear techniques. As has been mentioned earlier, unless checked "code growth" or "complexity drift" could even make it impossible to evolve useful codes. Hence, the problem needs understanding and control, for GP, which otherwise is a promising mechanism to solve a multitude of complex problems, to remain a viable technique.

References

[1] Byoung-Tak Zhang and Heinz Muhlenbein. *Balancing Accuracy and Parsimony in Genetic Programming*. Evolutionary Computation, 3(1): 17-38, 1995.

[2] Chris Gathercole and Peter Ross. *An Adverse Interaction Between Crossover and Restricted Tree Depth in Genetic Programming* - presented at GP 96

[3] John R.Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*.MIT Press, 1992.

[4] Justinian P. Rosca. *Generality Versus Size in Genetic Programming.* Proceedings of the Genetic Programming 1996 Conference (GP-96), The MIT Press.

[5] Justinian P. Rosca. *Analysis of Complexity Drift in Genetic Programming*. Proceedings of the Second Annual Conference on Genetic Programming, 1997.

[6] Lee Altenberg. *Emergent Phenomena in Genetic Programming.* Evolutionary Programming – Proceedings of the Third Annual Conference, pp233-241. World Scientific Publishing, 1994.

[7] Nicholas F. McPhee and J. D. Miller. *Accurate Replication in Genetic Programming.* Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95), pp303-309. Morgan Kaufmann.1995.

[8] Peter J. Angeline. *Subtree Crossover Causes Bloat.* Genetic Programming 1998: Proceedings of the Third Annual Conference, pp745-752, Wisconsin. Morgan Kaufmann.1998.

[9] Peter Nordin and Wolfgang Banzhaf. *Complexity Compression and Evolution.* ICGA95, pp310-317, Morgan Kaufmann. 1995.

[10] Peter Nordin, Frank Francone, and Wolfgang Banzhaf. *Explicitly Defined Introns and Destructive Crossover in Genetic Programming.* Advances in Genetic Programming 2, pp111-134. MIT Press, 1996.

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

[12] Tobias Blickle and L. Thiele. *Genetic Programming and Redundancy.* Proc. Genetic Algorithms within the Framework of Evolutionary Computation (Workshop at KI-94), Saarbrücken, Germany, 1994.

[13] Terence Soule, J. A. Foster, and J. Dickinson. *Code Growth in Genetic Programming.* Genetic Programming 1996: Proceedings of the First Annual Conference, pp215-223, Stanford. MIT Press.1996.

[14] Terence Soule and James A. Foster. *Code Size and Depth Flows in Genetic Programming.* Genetic Programming 1997, pp313-320. Morgan Kaufmann.1997.

[15] Terence Soule. *Code Growth in Genetic Programming. PhD. Dissertation.* University of Idaho. May 15,1998.

[16] Tobias Blickle. *Evolving Compact Solutions in Genetic Programming: A Case Study.* Parallel Problem Solving From Nature IV. LNCS 1141, pp564-573. Springer.1996.

[17] Sean Luke. *Issues in Scaling Genetic Programming: Breeding Strategies, Tree Generation, and Code Growth.* PhD. Thesis, 2000.

[18] William. B. Langdon, T. Soule, R. Poli, and J. A. Foster. *The Evolution of Size and Shape.* Advances in Genetic Programming 3, pp163-190. MIT Press.

[19] William. B. Langdon and R. Poli. *Fitness Causes Bloat: Mutation.* EuroGP '98, Springer-Verlag.1998.

[20] Willam.B. Langdon. *Quadratic Bloat in Genetic Programming.* Presented at GECCO'2000.