Crossover and Bloat in the Functionality Model of Enzyme Genetic Programming

Michael A. Lones and Andy M. Tyrrell
Bio-Inspired and Bio-Medical Engineering
Department of Electronics
University of York, UK
{Michael.Lones, Andy.Tyrrell}@bioinspired.com

Abstract— **The functionality model is a new approach in enzyme genetic programming which enables the evolution of variable length solutions whilst preserving local context. This paper introduces the model and presents an analysis of crossover and the evolution of program size.**

I. Introduction

In standard genetic programming (GP), the structure of a program is defined by a parse tree. The context of a component within a program, its relationship with other components, is given by its position within the parse tree.

Recombination is an important genetic operator and, in standard GP, takes the form of sub-tree crossover. Sub-tree crossover randomly selects a sub-tree from each parent parse tree and swaps them, creating child solutions. In each child, the component directly above a crossover point now receives one of its inputs from a different sub-tree. Given that sub-trees are chosen non-deterministically, in most cases this will lead to the component receiving input from a substantially different source to the one it received input from in the parent. In effect, the component's original context has been lost.

Loss of context during crossover is thought to be a significant reason why recombination in GP seldom generates viable offspring. One approach to improving the success of recombination is to select sub-trees deterministically, so that only sub-trees with similar function or topology are exchanged [1]. These context-preserving crossovers have the advantage of maintaining a parse tree representation but the disadvantage of a less natural, and more complicated, crossover operator.

A problem related to, and perhaps aggravated by, crossover is bloat. Bloat occurs when programs become larger and larger without significant improvement in function. In standard GP, where program bloat has nearly quadratic complexity [2], bloat can be controlled by placing limits on solution size or introducing a size penalty into the fitness function. However, both of these approaches modify or constrain the behaviour of search. The exact causes of bloat are not known, though a number of theories have been proposed. These include hitchhiking [3], protection from disruptive operators [4], operator biases [5], removal biases [6] and search space bias [7].

Enzyme genetic programming is an approach to genetic programming that uses a biomimetic representation for defining programs. This paper explains how this representation can be adapted to maintain local context during crossover and presents an analysis of the behaviour of crossover and the evolution of program size within enzyme GP. Section II describes enzyme GP. Section III introduces a new approach that enables a component to capture its own context independently of its environment. Section IV discusses program evolution and introduces a low-disruption form of crossover unique to enzyme GP. Section V analyses crossover performance and program size evolution. Section VI concludes.

II. Enzyme Genetic Programming

The representation of enzyme GP is based upon a simple model of metabolic pathways, the emergent structures that describe interactions between enzymes in cells. A description of this model, and the biological motivation behind enzyme genetic programming, can be found in [8] and [9].

From a non-biological perspective, enzyme GP represents a program as a collection of components where each component carries out a function and interacts with other components according to its own locally-defined interaction preferences. A program component is a terminal or function instance wrapped in an interface which determines both how the component appears to other components and, if the component requires input, which components it would like to receive input from. The fundamental principle behind enzyme GP is that the structure of a program is not given explicitly but is derived from connection choices made by each component of the program in a bottom-up, emergent, fashion.

A program is defined by a linear genotype where each component is encoded as a gene specifying its function and interaction preferences. Component types are named after the biological components whose behaviours they resemble. There are three types of component: glands, enzymes and receptors; which contain, respectively, input terminals, function instances and output terminals. For the following discussion, glands and receptors can be considered simple enzymes which either do not produce outputs

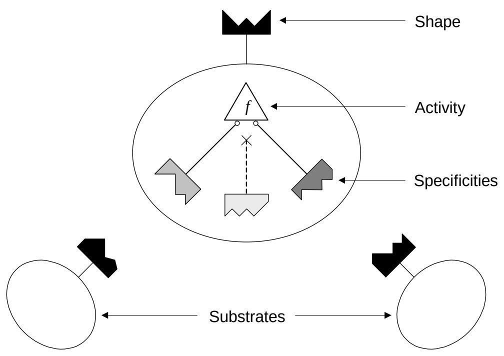


Fig. 1. Enzyme model. An enzyme consists of an activity and a set of specificities. Shape describes how the enzyme is seen by other program components. Specificity determines, in terms of their shape, which components will be bound as substrates (inputs).

(glands) to or do not receive inputs (receptors) from other components in the program.

The parts of an enzyme are depicted in figure 1. Activity is the function that the enzyme performs, or, viewing the enzyme as an interface, the function that it wraps. Specificities are templates which determine the interaction preferences of the enzyme. There is at least one specificity attached to each of the function's inputs (recessive specificities are discussed in [8] and [9]). During execution, these determine which other components each input will be received from; or more exactly, they specify the shape of the component that they wish to receive input from.

Shape is an identifier which describes how an enzyme is seen by other enzymes. Consequently, it is shape which determines how enzymes interact during execution and whose definition underlies the behaviour of enzyme GP. Previous implementations have used a definition of shape based upon activity, a definition which can not easily be applied to the evolution of variable-length programs. Functionality, introduced in the next section, is a new definition of shape designed to overcome both this and other problems.

The process of program development is independent of the exact definition of shape. An example is shown in figure 2. Development begins with *expression* of the receptors, which then choose substrates whose shapes are most similar to their specificities. Substrates are chosen from those defined in the genotype and can be either enzymes or glands. These substrates are now considered expressed and, if they require inputs, attempt to satisfy them by binding their own substrates. This process continues in hierarchical fashion until all expressed receptors and enzymes have satisfied all of their inputs. Each receptor is expressed exactly once, whilst glands and enzymes can be expressed zero or more times. Consequently, not all the components defined in a genotype need be expressed in the developed program and components are only expressed when they are compatible with other components that have already been expressed.

III. A Functionality Model

A component's shape is the means by which it is referenced by other components. Shape is both an identifier and a descriptor, and a good definition of shape should both distinguish between different components and capture what a component does. Ideally, it should describe a component's expected role in any program within which it is found, describing both the function it will perform and, if it takes inputs, the substrates to which it will apply this function. Functionality is a definition of shape which attempts to fulfill all these demands.

Functionality space is a vector space with a dimension for each member of the set of available functions and terminals. A functionality is a vector with a component between 0 and 1 for each member of this set. An example functionality space is depicted in figure 3.

The functionality of an enzyme is a weighted profile of the functions that occur in its ideal subtrees. An enzyme's subtrees are the hierarchical arrangements of substrates found below (and bound to) the enzyme's inputs. An ideal subtree occurs when the enzyme, and all other enzymes in the subtree, bind substrates that exactly match their specificities.

The functionality, F, of an enzyme is defined

$$(1-k)F(activity) + k\frac{\sum_{i=1}^{n} specificity_i}{n} \qquad (1)$$

where k is a constant called the input bias and n is the number of specificities. The functionality of the enzyme's activity, $F(activity)$, is a unit vector situated on the axis corresponding to the enzyme's function. The functionality of a gland is given by the functionality of its activity, since it has no specificities. Receptors do not have a functionality since they are never referenced by other components. An example of using equation 1 is illustrated in figure 4.

Accordingly, with shape defined as functionality, an enzyme's shape is derived from both its own activity and the

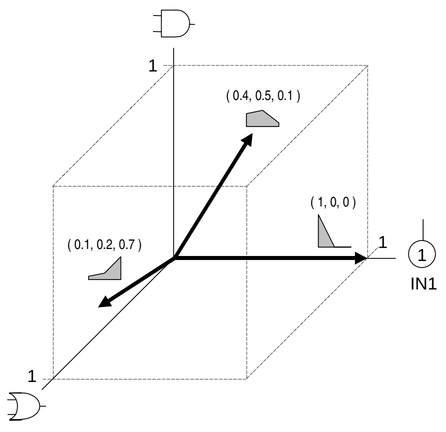


Fig. 3. Functionality space for function set {AND, OR} and terminal set {IN1} showing example functionalities. Vector plots of functionalities are to be used for illustrative purposes in this paper.

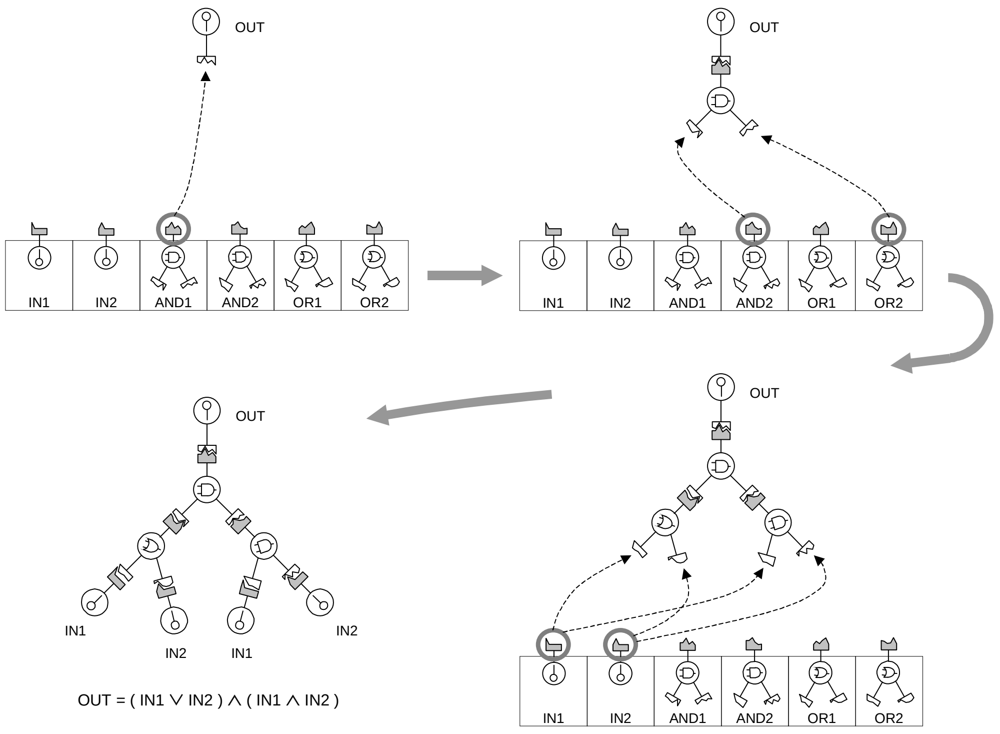


Fig. 2. Development of a program from a simple genotype. The output terminal receptor binds an AND1 enzyme as its substrate. The AND1 enzyme now chooses its own substrates and the process continues until the inputs of all expressed enzymes have been satisfied. Note that OR1 is never expressed and IN1 and IN2 are both expressed twice.

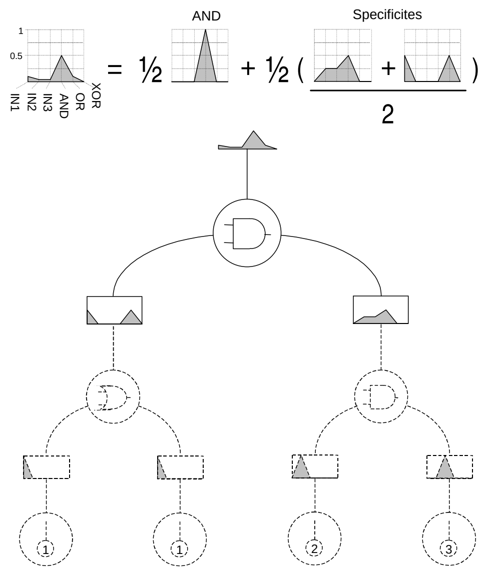


Fig. 4. Derivation of functionality. An AND enzyme's functionality is derived from the functionality of the AND function and its specificities using equation 1 with $k = \frac{1}{2}$.

shape of its specificities. Since its ideal substrates have the same shape as these specificities, the enzyme's shape also captures the shape of its ideal substrates and, following this logic recursively, the shape of all the components in its ideal subtrees.

However, functionality only captures a profile of the functions, weighted by depth, within an enzyme's ideal subtrees. It does not capture the hierarchical structure of the trees. Consequently, a functionality does not describe an enzyme uniquely and therefore specificity is not uniquely specific; much as a binding region of a biological enzyme only captures part of its substrate's shape. Nevertheless functionality space is continuous, making it unlikely that two non-identical enzymes will both have the same functionality and occur in the same program (an event which would still have precedent in biology).

IV. Evolution

Genotype evolution occurs within a spatially-distributed parallel genetic algorithm, details of which can be found in [8]. Bounded-size genotypes are constructed randomly to fill the initial population. New genotypes are created through crossover and mutation of existing genotypes.

Crossover in enzyme GP is somewhat different to subtree crossover in conventional GP and takes advantage of the fact that added components need not replace existing components. In subtree crossover, one subtree is always replaced by another. In enzyme GP crossover, a contiguous group of components is copied from one solution to another without removing any existing components (with the exception of receptors, see fig. 5). It is then up to other components within the program whether or not they use these new components. This has the ad-

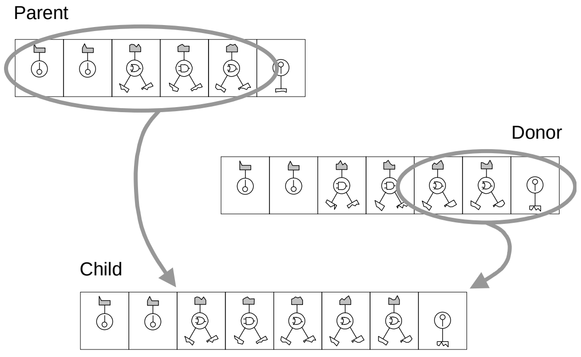


Fig. 5. A child genotype is a copy of a parent genotype augmented by a contiguous sequence of genes taken from a donor genotype. The number of receptors in a genotype is fixed, so any receptors transferred from the donor replace those copied from the parent.

vantage that there is no need to select which components will be replaced, avoiding the problems of either replacing components randomly, which is most likely to be meaningless, or replacing components deterministically, which can be inaccurate and expensive. In effect, this decision is made automatically by the rest of the program with new components subsuming the roles of existing components only if their shapes are preferred by other components. Only contiguous groups of components are copied between solutions, the intention being to encourage formation of building blocks where tightly-coupled genes encode tightly-coupled products.

The transfer operation is complemented by a converse operation which removes contiguous groups of components from solutions. Each of these operations is used, non-deterministically, for 50% of crossover events. The effect of the remove operation is to balance solution sizes so that recombination has an overall neutral effect upon solution size within a population. For both transfer and remove operations, the number of components targeted is chosen randomly within an upper limit. Neither operation can modifiy the internal contents of a component. Transfer followed by remove has an effect similar to a conventional crossover but with the added and removed components selected independently.

For comparison, enzyme GP has also been implemented with uniform crossover and headless crossover [10]. Headless crossover is identical to the crossover described above but with the second parent generated at random for each crossover event. The purpose of headless crossover is to give a comparison between crossover and mutation when implemented with the same mechanism, and to determine how well recombination exploits existing information.

Point-wise mutation is also used within enzyme GP, but to allow the behaviour of crossover to be measured, only targets numerical values such as the components of specificities. It is not currently used to change which function a component implements.

V. Analysis

Enzyme GP has been applied to the evolution of simple non-recurrent digital circuits. Here, an analysis is made of its performance and behaviour when evolving two-bit multiplier circuits. A two-bit multiplier is a function that takes two two-bit numbers as input and generates their product as a four-bit output. The fitness landscape of this function has been analysed by Vassilev [11], who found it to have a structure more suited for traversal by mutation than uniform crossover. In a sense, this makes the problem hard for enzyme GP which relies heavily on crossover and has a weak mutation operator. Results with this problem can be compared to the earlier activity model of enzyme GP [9] and, to a limited extent, Cartesian GP [12], which uses a graph-based representation. Unfortunately, results for this problem are not available for a tree-based GP.

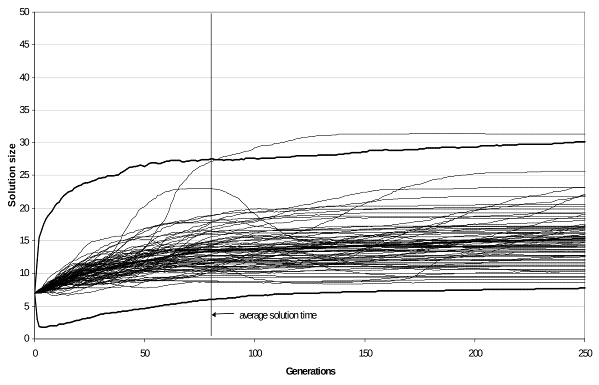
A. Performance of operators

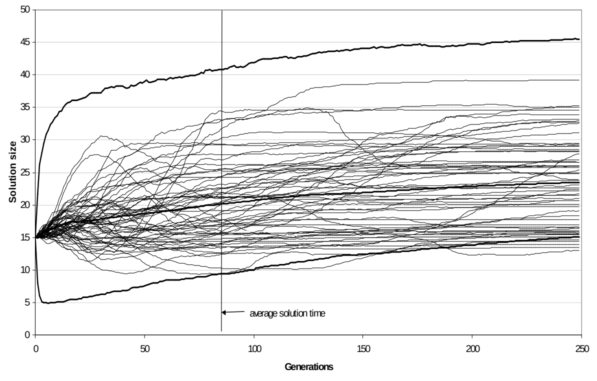
TABLE I
Performance on the two-bit multiplier problem.

Operators	Success	Generations
Standard c/o + mutation	69%	72
Uniform c/o + mutation	60%	104
Headless c/o + mutation	63%	98
Standard c/o only	0%	—
Headless c/o only	41%	49
Mutation only	65%	84
Activity model	55%	70

Table I shows average success rates and generation counts for different operators when applied to the multiplier problem with a population of 18x18. It is evident that standard enzyme GP crossover performs better than headless chicken crossover, which in turn performs better than uniform crossover. Mutation is quite capable by itself, but is improved by standard crossover. Without mutation, standard crossover makes no progress.

The reason standard enzyme GP crossover improves performance whereas uniform crossover does not stems from the fact that uniform crossover is very disruptive. In nature, uniform crossover is effective because the genotypes of a species have homologous structure and little variance in content. In artificial evolution, however, many solutions within a population are highly dissimilar and cutting and splicing them together will most likely produce children of lesser fitness. In standard enzyme GP crossover it is recognised that whilst two parents will most likely have significant differences, there may well be homologous regions. Unlike uniform crossover, this form of crossover stands some chance of copying one of these regions, and even if it does copy something unsuitable, the child solution does not have to use the new components. It may even later transfer these components to another solution where they are suitable.





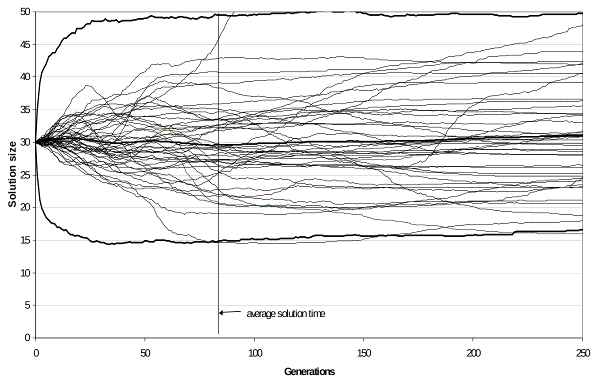


Fig. 6. Evolution of genotype size, with initial sizes 7, 15 and 30. Feint lines show average solution sizes for each run. Heavy lines indicate average minimum, average and maximum solution sizes across all runs. Transfer size $\leqslant 7$ for initial size 15 and $\leqslant 5$ for others.

Table I also shows how the functionality model compares to the earlier activity model. For the activity model, the solution is fixed length and the exact components are given in advance, making the problem easier. Success rate, however, is higher for the functionality model and solution time is comparable. Both models have a similar order of performance to cartesian GP (see [9]).

B. Evolution of solution size

Figure 6 shows the evolution of genotype size in enzyme GP for initial genotype sizes of 7, 15 and 30. Enzyme expression is about 65% at the start of a run and these initial genotype sizes concord with solutions below, at and above the minimum size for an optimal phenotype. Some genetic growth occurs for the first two cases, but tails off once most population members are above the minimum viable size. No net growth occurs in the third case where solution sizes are well above this level. Note that these observations reflect the average length at each generation. Individual runs often experience phases of expansion and compression before they reach a steady size.

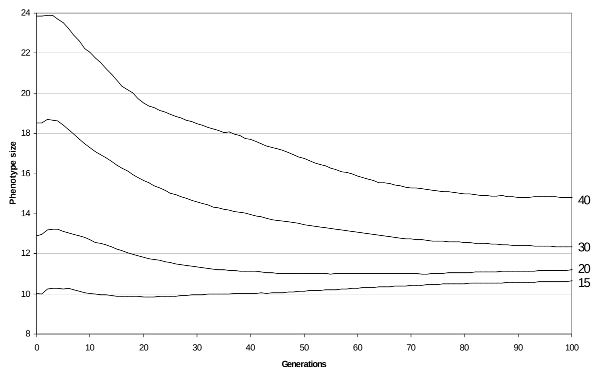


Fig. 7. Phenotype size evolution. Labels are initial genotype size.

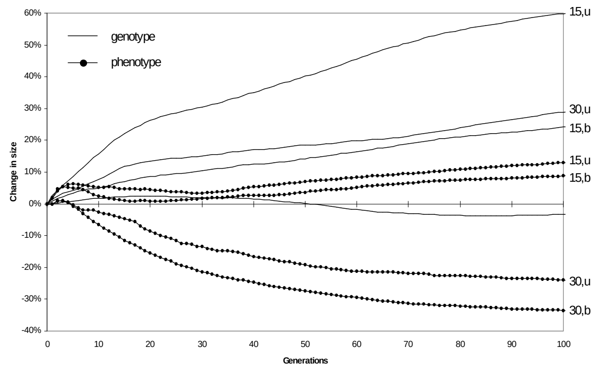


Fig. 8. Rates of growth in genotype and phenotype. Initial genotype sizes are 15 or 30. Recombination is (b)alanced or (u)nbalanced. Unbalanced recombination is 60% transfer, 40% remove.

Figure 7 shows the evolution of phenotype size for various initial genotype sizes. Figure 8 shows how phenotype size growth is related to genotype size growth for different starting and operator conditions. Where genotype growth occurs, phenotype growth occurs at a considerably lower level. For a starting size of 30, where there is no genotype growth, phenotype growth is negative. In the cases where recombination is biased to favour increase in size, genotype growth is considerably higher yet phenotype growth increases only slightly.

Since, on average, genotype size does not increase unless either it needs to or it is forced, enzyme GP does not suffer from bloat in the usual sense of the word. However, since genotype size remains constant whilst phenotype size decreases, it does suffer growth in the ratio of non-coding components to coding components within a genotype. This is interesting, since it is unusual for a variable-length representation not to bloat and perhaps more unusual for its coding component to experience negative growth. It seems that either there is some attractor that pulls phenotypes towards small sizes, but not too small, or there is an evolutionary advantage to having a greater non-coding content, yet not an advantage towards increasing genotype size.

The stability of solutions, the linkage between specificities and shapes, is yet to be analysed. However, it seems possible that smaller solutions could have greater stability than larger solutions, since larger solutions are more likely to have weak links—large distances between specificity and shape. If this is true, it would be less likely that they could be copied faithfully during crossover and propagate, either fully or in part, to future generations. Consequently, there would be an evolutionary advantage to smaller solutions, so long as they are large enough to compete in terms of fitness.

It is known that the presence of introns, non-coding components, can be beneficial to evolution [13]. At a simple level, introns can separate genes so that they are less likely to be the targets of crossover points. For this reason, introns are sometimes inserted intentionally into evolving solutions. In GP, introns also enable solutions to protect themselves from the disruptive effects of crossover by increasing the chance that crossover points will fall within unused sub-trees.

Miller has suggested another reason why introns may be beneficial to GP [14]. Cartesian GP is a form of GP which, like enzyme GP, has a genotype-phenotype mapping that does not require every component of the solution to be expressed. When cartesian GP is modified so that genotype length can change freely, genotype size is found to bloat, yet phenotype size does not bloat. Miller suggests that neutral exploration is one reason bloat occurs in standard GP; for most neutral variants will be longer than existing solutions. In cartesian GP, however, Miller speculates that most neutral variants are a result of changes to non-coding components of the solution, and lead to no increase in phenotype length.

If correct, lower stability of larger solutions and Miller's theory of bloat help explain why phenotype size decreases whereas genotype size does not. However, it is not yet clear why genotype size, so long as it is sufficiently high, does not suffer bloat in the same manner as other approaches to GP. This is likely to be the focus of further investigation.

VI. Conclusions

A new concept of shape, called functionality, has been introduced to enzyme genetic programming. Functionality allows the interaction preferences of a program component to be be described independently of the program it finds itself in. This, in turn, allows the evolution of variable length programs and preserves local context during crossover.

Analysis of program size evolution has shown that enzyme GP does not suffer from solution bloat in the manner of conventional GP. So long as solution size is large enough to solve the problem, average genotype size remains fairly constant. If solution size is too small, then genotype growth occurs. However, there is a marked tendency for small phenotypes to be selected over longer phenotypes, producing negative phenotype size growth. Since genotype size remains constant, this also leads to growth in the ratio between the proportion of non-coding and coding components.

The growth characteristics of enzyme GP have the advantage that smaller, and therefore more efficient, programs will be evolved. However, for problems where high-fitness sub-optima have short lengths compared to the optimum, this could increase the incidence of premature convergence.

References

[1] W. B. Langdon, "Size fair and homologous tree genetic programming crossovers," *Genetic programming and evolvable machines*, vol. 1, no. 1/2, pp. 95–119, April 2000.

[2] W. B. Langdon, "Quadratic bloat in genetic programming," in *Proceedings of the 2000 Genetic and Evolutionary Computation Conference*, D. Whitley et al., Ed. 2000, pp. 451–458, Morgan Kaufmann.

[3] W. A. Tackett, *Recombination, Selection, and the Genetic Construction of Computer Programs*, Ph.D. thesis, University of Southern California, Electrical Engineering Systems, 1994.

[4] T. Blickle and L. Thiele, "Genetic programming and redundancy," in *Genetic Algorithms within the Framework of Evolutionary Computation (Workshop at KI-94, Saarbrücken)*, J. Hopf, Ed., 1994, pp. 33–38.

[5] L. Altenberg, "Emergent phenomena in genetic programming," in *Evolutionary Programming – Proceedings of the Third Annual Conference*, A. V. Sebald and L. J. Fogel, Eds. 1994, pp. 233–241, World Scientific Publishing.

[6] T. Soule, J. A. Foster, and J. Dickinson, "Code growth in genetic programming," in *Genetic Programming 1996: Proceedings of the First Annual Conference*, J. R. Koza, Ed. July 1996, pp. 215–213, MIT Press.

[7] W. B. Langdon and R. Poli, "Fitness causes bloat," in *Soft Computing in Engineering Design and Manufacturing*, P. K. Chawdhry et al., Ed. 1997, pp. 13–22, Springer.

[8] M. A. Lones and A. M. Tyrrell, "Enzyme genetic programming," in *Proceedings of the 2001 Congress on Evolutionary Computation*. May 2001, vol. 2, pp. 1183–1190, IEEE Press.

[9] M. A. Lones and A. M. Tyrrell, "Biomimetic representation in genetic programming," in *Proceedings of the 2001 Genetic and Evolutionary Computation Conference Workshop Program*, July 2001, pp. 199–204.

[10] P. Angeline, "Subtree crossover: Building block engine or macromutation?," in *Genetic Programming 1997: Proceedings of the Second Annual Conference, GP97*, J. Koza et al, Ed. 1997, pp. 240–248, Morgan Kaufmann.

[11] V. Vassilev, J. Miller, and T. Fogarty, "On the nature of two-bit multiplier landscapes," in *The First NASA/DoD Workshop on Evolvable Hardware*, A. Stoica, D. Keymeulen, and J. Lohn, Eds. July 1999, pp. 36–45, IEEE Computer Society.

[12] J. Miller and P. Thomson, "Cartesian genetic programming," in *Third European Conference on Genetic Programming*, R. Poli et al, Ed. 2000, vol. 1802 of *Lecture Notes in Computer Science*, Springer.

[13] J. R. Levenick, "Inserting introns improves genetic algorithm success rate: Taking a cue from biology," in *Proceedings of the Fourth International Conference on Genetic Algorithms*, R. Belew and L. Booker, Eds. 1991, pp. 123–127, Morgan Kaufmann.

[14] J. Miller, "What bloat? Cartesian genetic programming on boolean problems," in *Proceedings of the 2001 Genetic and Evolutionary Computation Conference, Late Breaking Papers*, July 2001, pp. 295–302.