# Neutral Variations Cause Bloat in Linear GP

Markus Brameier and Wolfgang Banzhaf

Department of Computer Science, University of Dortmund
44221 Dortmund, Germany
{markus.brameier,wolfgang.banzhaf}@cs.uni-dortmund.de

**Abstract.** In this contribution we investigate the influence of different variation effects on the growth of code. A mutation-based variant of linear GP is applied that operates with minimum structural step sizes. Results show that neutral variations are a direct cause for (and not only a result of) the emergence and the growth of intron code. The influence of non-neutral variations has been found to be considerably smaller. Neutral variations turned out to be beneficial by solving two classification problems more successfully.

## 1 Introduction

One characteristic of genetic programming (GP) is that variable-length individuals grow in size. To a certain extent this growth is necessary to direct the evolutionary search into regions of the search space where sufficiently complex solutions with a high fitness are found. It is not recommended, in general, to initiate the evolutionary algorithm already with programs of a very large or even maximum size.

However, by the influence of variation operators and other reasons discussed in this paper genetic programs may grow too fast and too large such that the minimum size of programs required to solve the problem is exceeded significantly. As a result, finding a solution may become more difficult. This negative effect of code growth, i.e., that programs emerge larger than necessary without corresponding fitness improvements became known as the *bloat effect*. Code growth has been widely investigated in the GP literature [9,5,12,10,13,11,14,2] (see below). In general, a high complexity of GP programs causes an increase of evaluation time and reduces the flexibility of evolutionary manipulations. Moreover, it is argued to lead to a worse generalization performance.

Most evolutionary computation (EC) approaches model the Darwinian process of natural selection and adaptation. Contrary to this theory, Kimura's [8] neutral theory considers the random genetic drift of neutral mutations as the main force of evolution. In EC such variations are argued to explore flat regions of the fitness landscape more widely while non-neutral variations exploit regions with (positive or negative) gradient information. Banzhaf [1] first emphasized the relevance of neutral variations in genetic programming. Yu and Miller [16] demonstrated that neutral variations are advantageous after extra neutral code has been explicitly included into a graph representation of programs. Better performance was found for a Boolean problem with neutral mutations than without.

It is well-known, that a high proportion of neutral code (also referred to as introns) in genetic programs may increase the probability for variations to become neutral. But which type of variation creates the intron code in the first place? In our linear GP approach we apply minimum mutations. We demonstrate experimentally that neutral variations almost exclusively represent a *direct cause* for the growth of intron code. The influence of different variation effects on code growth and on prediction quality is verified for two approximation and two classification problems. Our observations differ from results reported for crossover-based GP which identify destructive variations as a direct [13] or indirect [12] cause of code growth.

## 2 Basics on Linear GP

In linear genetic programming (LGP) [3,6] the program representation consists of variable-length sequences of instructions from an imperative programming language. *Operations* manipulate variables (*registers*) and constants and assign the result to a destination register, e.g., $r_i := r_j + 1$. Single operations may be skipped by preceding *conditional branches*, e.g., $if(r_j > r_k)$.

The imperative program code is divided into *effective* and *non-effective* instructions. Such a separation of instructions already results from the linear program structure – prior to execution – and can be computed efficiently in linear runtime $O(n)$ where $n$ is the program length [6]. Only the effective code may influence program behavior. Non-effective instructions manipulate registers not impacting the program output at the current position and are, thus, not connected to the data flow generated by effective instructions. Non-effective instructions are also referred to as *structural introns* for a better distinction from *semantic introns* that may still occur within the (structurally) effective part of code [6]. For instance, all instructions preceding $r_0 := r_1 - r_1$ that only influence the content of register $r_1$ are semantic introns. Note that structural introns do not exist in tree-based GP, because in a tree structure, by definition, all program components are connected to the root. Hence, intron code in tree programs is semantic.

The length of a linear genetic program is measured as the number of instructions it holds. In linear GP the *absolute program length* and the *effective program length* are discerned. While the first simply includes all instructions of a program, the latter counts effective instructions only.

### 2.1 Variation Effects

Basically, two different effects of a variation operator can be discerned in EC. These are its effect on the genotype representation and its effect on the phenotype (fitness). In the current study, we focus on the proportion of constructive, destructive, and neutral operations per generation as semantic measurements of variation effects. If we assume that a better fitness always means a smaller fitness value the following definitions are valid: A variation is *constructive* if the difference in fitness between the parent individual $\mathcal{F}_p$ and the its offspring $\mathcal{F}_o$ is positive, i.e., $\mathcal{F}_p - \mathcal{F}_o > 0$. In case of a negative difference we refer to a *destructive* variation, i.e., $\mathcal{F}_p - \mathcal{F}_o < 0$. Finally, a genetic operation is *neutral* if it does not change the fitness, i.e., $\mathcal{F}_p = \mathcal{F}_o$.

On the structural level we measure the proportion of effective and non-effective variations. According to the distinction between effective code and non-effective code, as defined above, let an *effective variation* denote a genetic operation that modifies the effective code of a linear genetic program. Otherwise, a variation is called *non-effective*. Note that there is no change of program behavior (fitness) guaranteed by such (structurally) effective variations.

The notion of *variation step size* refers to the *amount* of structural change between parent and offspring that is induced by the variation operator. In this paper we apply a pure mutation-based variant of linear GP that induces minimum variation steps on the imperative program structure. We distinguish macro-mutations from micro-mutations. Programs grow by *macro-mutations* which include insertions or deletions of single random instructions. *Micro-mutations* exchange the smallest program components that comprise a single operator, a register or a constant.

## 3 Code Growth in GP

Several theories have been proposed to explain the phenomenon of code bloat in genetic programming. Basically, three different causes of code growth have been distinguished up to now that do not contradict each other while each being capable of causing code growth for itself. In general, the minimally required complexity of a solution may be exceeded by incorporating intron code (may be removed without changing the program behavior) or by mathematically equivalent extensions. All causes require the existence of fitness information, i.e., may not hold on (completely) flat fitness landscapes. The (effective) program size develops depending on how strongly it is correlated to the fitness. In this way, fitness may be regarded as a necessary precondition for code growth.

One theory (*protection theory*) [12,5,2,14] argues that code growth occurs as a protection against the destructive effects of crossover. The destructive influence on the program structure strongly depends on the absolute variation step size. If the maximum amount of code that may be exchanged in one variation step is large, e.g., restricted only by the program size, evolution may reduce the strength of variation on the effective code by developing a higher proportion of introns within the replaced subprograms. This phenomenon may occur when using crossover as well as subprogram mutations.

Another theory (*drift theory*) [10,11] claims that code growth results from the structure of the search space or, more precisely, from the distribution of semantically identical solutions. For many problems more larger program solutions exist with a certain fitness than smaller ones. Therefore, larger solutions are created and selected for a higher probability.

Finally, the third theory (*bias theory*) [15,11,14] of code growth is based on the hypothesis of a removal bias in tree-based GP. The potential destruction caused by removing a subtree depends on the subtree size. The effect of the replacing subtree on the fitness, instead, is independent from its size. As a results, the growing offspring from which the smaller subtree is removed (and in which the longer is inserted) will survive for a higher probability than the shrinking offspring.

Soule *et al.* [13] demonstrated for tree-based GP that significantly less code growth (especially of introns) emerges if only those offsprings are incorporated into the population that perform better than their parents. The authors hold the missing destructive crossover results responsible for this behavior. While a direct influence of destructive variations on the growth of (intron) code is not doubted here, it has to be noted that not only destructive but also neutral variations are excluded from evolutionary progress in [13]. Moreover, the proportion of (the remaining) constructive variations is usually rather low in GP.

If we want to clearly identify a reason for code growth it is important to design the experiment in such a way that the other mechanisms (if existent) are disabled as much as possible. In linear GP, the protection theory may not be valid if the step size of the variation operator is reduced to a minimum and code is not exchanged, but only added *or* removed. Both may be achieved easily for the imperative program structure by single instruction mutation as described above.

With a mutation step size of one instruction only, intron instructions cannot be inserted or deleted directly along with a *non*-neutral variation. In particular, this allows destructive variations to be analyzed with only a minimum influence on the size of intron code. Structural introns may only emerge with such operations by deactivation of other depending instructions (apart from the mutation point). The same is true for the creation of introns on the semantic level. In general, linear GP allows structural variation steps to be permanently minimum at each position of the genom. On reason for this is that the data flow in linear genetic programs is graph-based [4]. Due to stronger constraints of the tree representation, small variation step sizes are especially difficult in upper tree regions. If single tree nodes are tried to be deleted only one of its subtrees may be reconnected while the others get lost.

The influence of the second cause is reduced, too, because the difference between parent and offspring is only one instruction. At least, using such minimum variation steps exclusively will make the evolutionary process drift less quickly towards more complex regions of the search space. In general, the maximum step size of a variation operator decides on the potential maximum speed of code growth but does not represent a explicit force (if the variation operator is not length-biased).

## 4 Conditional Variation

We use a steady state evolutionary algorithm that applies tournament selection with a minimum of two participants per tournament. Variations happen on copies of the parent individuals (tournament winners) that replace the tournament losers. The integration of newly created individuals into the population is restricted so that offsprings are accepted only if they result from certain types of variation (see Section 2.1). Such a *conditional acceptance* of a variation implies automatically that the reproduction of parents is omitted, too, since the population remains unchanged.

## 5 Benchmark Problems

The different experiments documented in this contribution are conducted with four benchmark problems – including two symbolic regressions and two classification tasks. The first problem is represented by the two-dimensional *mexican hat* function as given by Equation 1. The function constitutes a surface in three-dimensional space that resembles a mexican hat.

$$f_{mexicanhat}(x,y) = \left(1 - \frac{x^2}{4} - \frac{y^2}{4}\right) \times e^{\left(-\frac{x^2}{8} - \frac{y^2}{8}\right)} \quad (1)$$

The second regression problem, called *distance*, requires the Euclidean distance between two points (vectors) $\boldsymbol{x}$ and $\boldsymbol{y}$ in $n$-dimensional space to be computed by the genetic programs (see Equation 2). The higher the dimension is chosen ($n = 3$ here) the more difficult the problem becomes.

$$f_{distance}(x_1, y_1, .., y_n, y_n) = \sqrt{(x_1 - y_1)^2 + .. + (x_n - y_n)^2} \quad (2)$$

The third problem is the well-known *spiral* classification [9] where two intertwined spirals have to be distinguished in two-dimensional data space. Finally, the *three chains* problem concatenates three rings of points that each represent a different data class. Actually, one "ring" denotes a circle of 100 points in three-dimensional space whose positions are slightly noisy. The rings approach each other at five regions without leading to intersection. The problem difficulty may be scaled up or down depending on both the angle of the rings to one another and on the number of rings.

## 6 Experimental Setup

Table 1 summarizes attributes of the data sets that have been created for each test problem. Furthermore, problem-specific configurations of our linear GP system are given that comprise the compositions of the function set, the fitness function, and the number of registers.

It is important for the performance of linear GP to provide enough registers for calculation, especially if the input dimension is low. Thus, the total number of available registers – including the minimum number that is required for the input data – is an important parameter. In general, the number of registers decides on the number of program paths that can be calculated in parallel. If it

**Table 1.** Problem-specific parameter settings.

| Problem | *mexican hat* | *distance* | *spiral* | *three chains* |
|---|---|---|---|---|
| Problem type | Regression | Regression | Classification | Classification |
| #Inputs | 2 | 6 | 2 | 3 |
| Input range | $[-4.0, 4.0]$ | $[0, 1]$ | $[-2\pi, 2\pi]$ | $[0, 5]$ |
| Output range | $[-1, 1]$ | $[0, 1]$ | $\{0, 1\}$ | $\{0, 1, 2\}$ |
| #Output classes | – | – | 2 | 3 |
| #Registers | 6 | 12 | 6 | 6 |
| #Fitness cases | 400 | 300 | 194 | 300 |
| Fitness function | SSE | SSE | CE | CE |
| Instruction set | $\{x^y\}$ | $\{\sqrt{x}, x^2\}$ | $\{sin, cos, if >\}$ | $\{x^y, if >\}$ |
| $\cup \{+, -, \times, /\}$ | | | | |

**Table 2.** General parameter settings.

| Parameter | Setting | Parameter | Setting |
|---|---|---|---|
| Number of generations | 1000 | Initial program lengths | 5-15 |
| Population size | 1000 | Macro-mutations | 75% |
| Tournament size | 2 | Micro-mutations | 25% |
| Maximum program length | 200 | Set of constants | $\{1, .., 9\}$ |

is not sufficient there may be too many conflicts by overwriting register content within programs.

For the approximation problems the fitness is defined as the continuous *sum of square errors* (SSE) between the predicted outputs and the example outputs. For the two classification tasks specified in Table 1 the fitness function is discrete and equals the *classification error* (CE) here, i.e., the number of wrongly classified inputs.

The *spiral* problem applies an *interval classification* method, i.e., if the output is smaller than 0.5 it is interpreted as class 0, otherwise it is class 1. For the *three chains* problem we use an *error classification* method, instead. That is the distance between the problem output and one of the given output classes (0, 1, or 2) must be smaller than 0.5 to be accepted as correct. General configurations of our linear GP system are summarized in Table 2 and are valid for all experiments and test problems.

## 7 Experimental Results

The experiments documented in Tables 3 to 6 investigate the influence of different variation effects on both, the complexity of (effective) programs and the prediction performance. The average prediction error is calculated by the best solutions of 100 independent runs together with the statistical standard error.

**Table 3.** *Mexican hat* problem: Conditional acceptance of mutation effects and conditional reproduction. Average results over 100 runs.

| Experiment ID | SSE | | Length | | | Variations (%) | | |
|---|---|---|---|---|---|---|---|---|
| | *mean* | *std.err.* | *abs.* | *eff.* | % | *constr.* | *neutral* | *noneff.* |
| std | 3.5 | 0.5 | 140 | 60 | 43 | 0.8 | 54 | 52 |
| nodestr | 3.3 | 0.5 | 139 | 61 | 44 | 0.2 | 53 | 52 |
| noneutr | 1.6 | 0.1 | **38** | **28** | 72 | 7.5 | 37 | 34 |
| nononeff | 1.5 | 0.1 | **41** | **30** | 74 | 4.8 | 41 | 32 |

**Table 4.** *Distance* problem: Conditional acceptance of mutation effects and conditional reproduction. Average results over 100 runs.

| Experiment ID | SSE | | Length | | | Variations (%) | | |
|---|---|---|---|---|---|---|---|---|
| | *mean* | *std.err.* | *abs.* | *eff.* | % | *constr.* | *neutral* | *noneff.* |
| std | 6.5 | 0.3 | 78 | 32 | 41 | 0.5 | 63 | 63 |
| nodestr | 8.0 | 0.3 | 78 | 32 | 41 | 0.1 | 64 | 63 |
| noneutr | 6.0 | 0.3 | **24** | **15** | 63 | 6.3 | 48 | 47 |
| nononeff | 6.5 | 0.2 | **25** | **16** | 62 | 4.7 | 52 | 48 |

The absolute and the effective program length are averaged over all programs that are created during runs. (Figure 1 shows exemplarily the generational development of the average program length in the population.) Due to the small step size of mutations used here, the average length of best individuals develops almost identically (not documented). The proportion of effective code is given in percent while the remaining proportion comprises the structural introns. Additionally, we calculate the average proportions of constructive, neutral and noneffective variations among all variations during a run (see Section 2.1). The rates of destructive and effective variations are obvious then.

In the no∗ experiments of Tables 3 to 6 offsprings are not inserted into the population if they result from a certain type of variation. Additionally, the reproduction of the parent individuals is skipped. Simply put, the variation is canceled completely without affecting the state of the population. Nevertheless, with all configurations the same number of variations (and evaluations) happens, i.e., the same number of new individuals (1000) defines a generation. Thus, unaccepted variations are still included in the calculation of the prediction error, the program lengths and the variation rates.

The standard mutation approach std is characterized by a balanced ratio of neutral operations and non-neutral operations, on the one hand, and effective operations and non-effective operations, on the other hand.

Destructive variations hardly contribute to the evolutionary progress here. The average prediction error changes only slightly with both the two continuous test problems, *mexican hat* and *distance*, and the two discrete test problems, *spiral* and *three chains*, if offsprings from destructive variations are not accepted

**Table 5.** *Spiral* problem: Conditional acceptance of mutation effects and conditional reproduction. Average results over 100 runs.

| Experiment ID | CE | | Length | | | Variations (%) | | |
|---|---|---|---|---|---|---|---|---|
| | *mean* | *std.err.* | *abs.* | *eff.* | % | *constr.* | *neutral* | *noneff.* |
| std | 13.6 | 0.6 | 128 | 64 | 50 | 0.3 | 50 | 42 |
| nodestr | 12.4 | 0.5 | 117 | 64 | 55 | 0.02 | 46 | 39 |
| noneutr | 20.0 | 0.6 | **37** | **31** | 82 | 5.0 | 32 | 20 |
| nononeff | 13.1 | 0.5 | 69 | 62 | 89 | 1.5 | 32 | 13 |

**Table 6.** *Three chains* problem: Conditional acceptance of mutation effects and conditional reproduction. Average results over 100 runs.

| Experiment ID | CE | | Length | | | Variations (%) | | |
|---|---|---|---|---|---|---|---|---|
| | *mean* | *std.err.* | *abs.* | *eff.* | % | *constr.* | *neutral* | *noneff.* |
| std | 15.5 | 0.6 | 132 | 57 | 43 | 0.2 | 62 | 49 |
| nodestr | 16.4 | 0.7 | 124 | 53 | 43 | 0.03 | 62 | 49 |
| noneutr | 24.6 | 0.8 | **34** | **28** | 82 | 5.3 | 38 | 20 |
| nononeff | 12.9 | 0.7 | 80 | 71 | 88 | 1.0 | 45 | 13 |

(nodestr). This is true even though about 50 percent of all variations are rejected and even if the rate of constructive variations decreases significantly, especially with the classification problems (in Tables 5 and 6). Hence, almost only neutral variations are responsible for evolution here. Obviously, the probability for selecting an individual, that performs worse than its parent, seems to be so low, on average, that it hardly makes any difference if this individual is copied into the population or not. Due to the low survival rate of these offsprings and due to the small mutation step size (see below), destructive mutations almost do not have any influence on code growth here.

The influence of neutral variations is in clear contrast to the influence of destructive variations. Obviously, the survival probability of offsprings is higher after a neutral (or a constructive) variation. This facilitates both a continuous further development of solutions and the growth of programs. An important result is that both the absolute size and the effective size of programs are reduced most if we skip neutral variations (noneutr).

*Non-effective neutral* variations, as defined in Section 2.1, create or modify non-effective instructions, i.e., structural introns. Accordingly, we may assume that mostly *effective neutral* variations are responsible for the emergence of semantic introns – within the (structurally) effective part of program. Effective neutral variations (and semantic introns) are harder to induce if the fitness function is continuous and, thus, occur less frequently. This is reflected here with the two regression problems by similar rates of non-effective operations and neutral operations. For the discrete classification problems, instead, the proportion of neutral variations has been found significantly larger than the proportion of non-effective variations which means a higher rate of effective neutral variations. Additionally, the frequency of neutral variations on the effective code depends on the function set. Especially, branches create semantic introns easily while the resulting larger effective code indirectly increases the probability for effective (neutral) variations.

In the nononeff experiments non-effective variations are rejected, i.e., only effective variations are accepted. This includes effective neutral variations in contrast to the noneutr experiment. Semantic introns created by those variations may be responsible for the larger effective code that occurs with both classifications in nononeff runs. With the two regressions the effective size is half-reduced for both noneutr and nononeff because most neutral variations are non-effective here.

We may conclude that neutral variations – in contrast to destructive variations – dominate code growth almost exclusively. Since mutation step sizes are small, constructive variations may only play a minor role for code growth already because of their low frequency. This is true even if the rate of constructions increases (together with the rate of destructions) when not accepting the result of neutral variations in the population (noneutr). One reason for this is the lower rate of structural and semantic introns. Moreover, non-neutral variations may hardly be responsible for an (unnecessarily) growth of code here because the variation step size is minimum. Then intron code cannot be directly created by such operations and *all* changes of a program are exposed to fitness selection.

As noted in Section 3, the possibility to induce small structural mutations at each position of the linear representation is important for our results. Indirect creation of intron instruction by deactivations seems to play a minor role only. Note that due to changing register dependencies non-effective (effective) instructions may be reactivated (deactivated) in a linear genetic program above the mutated instruction. Besides, an increasing robustness of the effective code lets deactivation of instructions occur less frequently in the course of a run [7].

When step sizes are larger, i.e., more than one instruction may be inserted per variation, as this occurs with crossover, programs may grow faster and by a smaller total number of variations. In particular, introns may be directly inserted by variations, too, that are not neutral as a whole.

Concerning the prediction quality the noneutr experiment has a small positive or no effect with the two approximation problems but a clear negative effect with the two classification problems. Contrary to this, the performance never drops in the nononeff experiment (compared to the baseline result). Consequently, effective neutral variations may be supposed to be more relevant than non-effective neutral variations, in general. This is not obvious, because all neutral changes may be reactivated later in (non-neutral) variations.

We may not automatically conclude here that neutral variations are more essential for solving classifications only because those problems are discrete. It has to be noted, that a better performance may also result from the fact that programs grow larger by neutral variations. Depending on the problem definition, the configuration of the instruction set, and the observed number of generations,

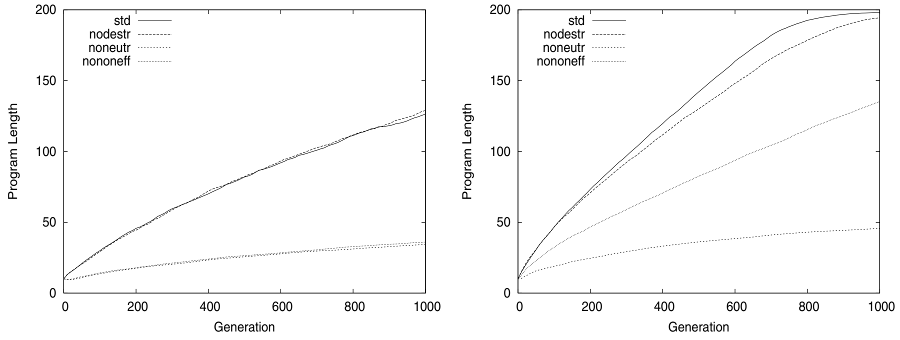

**Fig. 1.** Development of absolute program length for *distance* (left) and *three chains* (right) (similar for *mexican hat* and *spiral*). Code growth significantly reduced without neutral variation effects. Average figures over 100 runs.

the optimum speed of code growth may be quite different. By making use of branches, that allow many special cases to be considered in a program, both classification problems profit less from a lower complexity of solutions than the two symbolic regressions.

## 8 Conclusion

We have analyzed the influence of different variation effects on the development of program size for a mutation-based LGP approach. In all test cases neutral variations have been identified as a major reason for code growth. Almost no bloat effect occurred if (1) neutral variations are not accepted and (2) the variation step size is reduced to a minimum. Especially, the linear (imperative) representation of programs allows structural variation steps to be constantly small.

## Acknowledgements

This research was supported by the German Research Community (DFG), Collaborative Research Center (SFB) 531, project B2.

## References

1. W. Banzhaf, *Genotype-Phenotype-Mapping and Neutral Variation: A Case Study in genetic programming.* In *Proceedings of the Conference on Parallel Problem Solving from Nature III*, pp. 322–332, Springer-Verlag, Berlin, 1994.
2. W. Banzhaf and W.B. Langdon, *Some considerations on the reason for bloat.* Genetic Programming and Evolvable Machines, vol. 3(1), 81–91, 2002.
3. W. Banzhaf, P. Nordin, R. Keller, and F. Francone, *Genetic Programming – An Introduction. On the Automatic Evolution of Computer Programs and its Application.* dpunkt/Morgan Kaufmann, Heidelberg/San Francisco, 1998.

4. W. Banzhaf, M. Brameier, M. Stautner, and K. Weinert. *Genetic Programming and its Application in Machining Technology.* In H.-P. Schwefel *et al.* (eds.) *Advances in Computational Intelligence – Theory and Practice*, Springer, Berlin, 2002.
5. T. Blickle and L. Thiele, *Genetic Programming and Redundancy.* In J. Hopf (ed.) *Genetic Algorithms within the Framework of Evolutionary Computation* (Workshop at KI-94), pp. 33–38, Max-Planck-Institut für Informatik, Technical Report No. MPI-I-94-241, 1994.
6. M. Brameier and W. Banzhaf, *A Comparison of Linear Genetic Programming and Neural Networks in Medical Data Mining.* IEEE Transactions on Evolutionary Computation, vol. 5(1), pp. 17–26, 2001.
7. M. Brameier and W. Banzhaf, *Explicit Control of Diversity and Effective Variation Distance in Linear Genetic Programming.* In J.A. Foster *et al.* (eds.) *Genetic Programming, Proceedings of the 5th European Conference (EuroGP 2002)*, pp. 37–49, Springer-Verlag, LNCS, Berlin, 2002.
8. M. Kimura, *The Neutral Theory of Molecular Evolution.* Cambridge University Press, 1983.
9. J.R. Koza, *Genetic Programming.* MIT Press, Cambridge, MA, 1992.
10. W.B. Langdon and R. Poli, *Fitness Causes Bloat.* In P.K. Chawdhry *et al.* (eds.) *Soft Computing in Engineering Design and Manufacturing*, pp. 13–22, Springer-Verlag, Berlin, 1997.
11. W.B. Langdon, T. Soule, R. Poli, and J.A. Foster, *The Evolution of Size and Shape.* In L. Spector *et al.* (eds.) *Advances in Genetic Programming III*, pp. 163–190, MIT Press, Cambridge, MA, 1999.
12. P. Nordin and W. Banzhaf, *Complexity Compression and Evolution.* In L.J. Eshelman (ed.) *Proceedings of the Sixth International Conference on Genetic Algorithms (ICGA'95)*, pp. 310–317, Morgan Kaufmann, San Francisco, CA, 1995.
13. T. Soule and J.A. Foster, *Code Size and Depth Flows in Genetic Programming.* In J.R. Koza *et al.* (eds.) *Genetic Programming 1997: Proceedings of the Second Annual Conference (GP'97)*, pp. 313–320, Morgan Kaufmann, San Francisco, CA, 1997.
14. T. Soule and R.B. Heckendorn, *An Analysis od the Causes of Code Growth in Genetic Programming.* Genetic Programming and Evolvable Machines, vol. 3(3), pp. 283–309, 2002
15. T. Soule and J.A. Foster, *Removal Bias: A new Cause of Code Growth in Tree-based Evolutionary Programming.* In *Proceedings of the International Conference on Evolutionary Computation (ICEC'98)*, pp. 781–786, IEEE Press, 1998.
16. T. Yu and J. Miller, *Neutrality and the Evolvability of Boolean Function Landscapes.* In J.F. Miller *et al.* (eds.) *Genetic Programming, Proceedings of the 4th European Conference (EuroGP 2001)*, pp. 204–217, Springer-Verlag, LNCS, Berlin, 2001.