Shorter Fitness Preserving Genetic Programs

Anikó Ekárt

Computer and Automation Research Institute
Hungarian Academy of Sciences
1518 Budapest, POB. 63, Hungary
E-mail: ekart@sztaki.hu

Abstract. In the paper a method that moderates code growth in genetic programming is presented. The addressed problem is symbolic regression. A special mutation operator is used for the simplification of programs. If every individual program in each generation is simplified, then the performance of the genetic programming system is slightly worsened. But if simplification is applied as a mutation operator, more compact solutions of the same or better accuracy can be obtained.

1 Introduction

An important problem with genetic programming systems is that in the course of evolution the size of individual programs is continuously growing. The programs contain more and more non-functional code. When evaluating the genetic programs over the fitness cases, much of the time is spent on these irrelevant code fragments. Thus, they reduce speed and, in the meantime, make the programs unintelligible for humans.

Koza [7] proposes the use of a maximum permitted size for the evolved genetic programs as parameter of genetic programming systems. Accordingly, genetic programs are allowed to grow until they reach a predefined size. In the same work a so-called editing operation is proposed for (1) making the output of genetic programming more readable and (2) producing simplified output or improving the overall performance of genetic programming. The editing operation consists in the recursive application of a set of simplifying rules. However, it was shown that for the boolean 6-multiplexer problem the application of the editing operation does not influence the performance of genetic programming in a notable way.

Hooper and Flann [5] apply expression simplification in a simple symbolic regression problem. They conclude that the accuracy of genetic programming could be improved by simplification. Additionally, simplification could (1) prevent code growth and (2) introduce new useful constants.

Langdon [8] introduces two special crossover operators, the so-called *s*ize fair and *h*omologous crossovers. These operators create an offspring by replacing a subtree of one parent with a carefully selected similar-size subtree of the other parent. By using these operators, code growth is considerably reduced without affecting the performance of genetic programming.

There are several studies that suggest taking into account the program size when computing the fitness value.

Iba, de Garis and Sato [6] define a fitness function based on a *Minimum Description Length* (MDL) principle. The structure of the tree representing the genetic program is reflected in its fitness value:

$$mdl = Error\_Coding\_Length + Tree\_Coding\_Length.$$

Zhang and Mühlenbein [16] demonstrate the connection between accuracy and complexity in genetic programming by means of statistical methods. They use a fitness function based on the MDL principle:

$$Fitness_i(g) = E_i(g) + \alpha(g)C_i(g),$$

where $E_i(g)$ and $C_i(g)$ stand for the error and the complexity of individual i in generation g. The Occam factor $\alpha(g)$ is computed as a function of the least error in the previous generation $E_{best}(g-1)$ and the estimated best program size for the current generation $\hat{C}_{best}(g)$. Thus, their fitness function is adaptively changing from generation to generation.

Soule, Foster and Dickinson [12] compare two methods for reducing code growth in a robot guidance problem: (1) the straightforward editing out of irrelevant and redundant parts of code and (2) the use of a fitness function that penalizes longer programs. They conclude that applying the penalty outperforms *any kind* of editing out, so providing new evidence for [6, 16].

Notwithstanding, other studies show that these seemingly irrelevant or redundant parts of code are useful because they shield the highly-fit building blocks of programs from the destructive effects of crossover.

Angeline [1] calls these apparently useless fragments of code *introns*, in analogy with the introns contained in DNA. He points out that the formation of introns should not be hindered, since they provide a better chance for the transfer of complete subtrees during crossover.

Banzhaf et al. [2] argue that the analogy to biological introns might be wrong. But since *intron* is already a common term in the genetic programming domain, we will use it throughout the paper with the meaning of *non-functional code.*

Nordin, Francone and Banzhaf [9] demonstrate through experiments that introns allow a population to keep the highly-fit building blocks and in the meantime make possible the protection of individuals against destructive crossover. They introduce the so-called *Explicitly Defined Introns* that are inserted in the code and serve as a control mechanism for the crossover probability of the neighboring nodes.

In the present paper we describe a method that takes both advice into account:

- Code growth in genetic programming should be limited in order to obtain a comprehensible solution in a reasonable amount of time.
- Introns should be preserved, since they shield the highly-fit building blocks from the harmful effects of crossover.

The paper is organized as follows: In Section 2 the biological evidence that inspired this work is presented. In Section 3 the method is described and in Section 4 the results of 800 runs of the system on two symbolic regression problems are shown. Then a real-world application using these results is discussed and conclusions are drawn.

2 Biological Background

The DNA of bacteria contains continuous coding sequences. For many years it was believed that the genes of higher organisms are also continuous. This view was changed in 1977, when it was discovered that the genes of some eukaryotic organisms (made of nucleated cells) are discontinuous, and the non-coding sequences are much longer than the coding sequences.

Generally, the genes of living organisms consist of:

- exons - base sequences that encode proteins or polypeptides; and
- introns - base sequences that do not participate directly in the production of proteins.

The introns can influence the amount and the quality of the proteins expressed by the gene in which they occur. But the mechanism of the indirect effect of introns on protein production is not known. They represent regions in which DNA can break and recombine without affecting the encoded proteins [13].

There is evidence suggesting that: *"introns were present in ancestral genes and were lost in the evolution of organisms that have become optimized for very rapid growth, such as eubacteria and yeast"* [13].

Gilbert [4] points out that *"introns have been used to assemble those genes that are the late product of evolution"*. At the same time, he brings evidence for the *loss* of introns during evolution.

Thus, introns play an important role in evolution, when shielding the exons from destruction through crossover. But they can disappear in the course of evolution.

On the other hand, the main source of variability is mutation. From the many existing types of mutation, we consider the following [2, 14, 15]:

– point mutation - change of one base pair to another;
– neutral mutation - a genetic change that is neither advantageous nor disadvantageous for the organism;
– frameshift mutation - insertion or deletion of one or more base pairs; and
– large DNA sequence rearrangement.

Usually, in genetic programming systems the analog of the first type is implemented. In addition, we also consider here the other mutation types.

3 The Proposed Method

The goal of this work is to reduce the size of genetic programs evolved in symbolic regression problems. However, the method could be applied to any genetic programming system after defining the corresponding rules that simplify the structure of programs.

We designed a special mutation operator that modifies only the structure of a genetic program; the interpretation and the fitness value remain the same (therefore, it could be seen as the analog of the biological neutral mutation). This modification is intended to eliminate the occasional introns and simplify the structure of the genetic program, without altering its accuracy (in a similar way to the editing operation of [7] and the expression simplifier of [5]). Since the problem is symbolic regression, this mutation operator performs the algebraic simplification of the expression of a genetic program. The simplifier is implemented in Prolog and consists of approximately 250 clauses. Some of the simplification rules are shown in Table 1.

Let us see two short examples:

$$f(x,y) = x*2 - x*(3-1) + 3*x/y \overset{\text{simpl}}{=} 3*x/y$$
$$g(x) = x*(x-2) + 3*x \overset{\text{simpl}}{=} x + x*x$$

In the first example the non-functional part was removed and in the second one algebraic simplification was performed.

One can see the analogy of this simplification to the biological mutation:

Table 1. Some simplification rules

Original expression	Simplified expression	Binding
$0 + x$	x	
$K_1 * x + K_2 * x$	$K * x$	$K = K_1 + K_2$
$K_1 * K_2 * x$	$K * x$	$K = K_1 * K_2$
$(-1) * x$	$-x$	

– simplification as a whole - neutral mutation;
– removal of non-functional code - frameshift mutation; and
– algebraic simplification - large DNA sequence rearrangement.

The removal of non-functional code is more restricted than frameshift mutation, since it is applied only to non-functional code and there is no addition, just deletion of this code.

The algebraic simplification is in fact more than a rearrangement. For a more precise analogy with biology we could have made just a simple transformation, such that:

$$g(x) = x * (x - 2) + 3 * x \stackrel{\text{transf}}{=} (3 * x - 2 * x) + x * x.$$

This transformation is closer to large DNA sequence rearrangement and could be the subject of later experiment.

Since the simplification of an algebraic expression involves the removal of non-functional code, we decided for a single "new" mutation operator, that performs all possible simplifications on the selected expression (like the editing operation proposed by Koza [7]).

We applied this mutation operator in addition to the usual recombination operators (crossover and point mutation). We thought that applying the simplification in every generation might be too drastic and time-consuming and, therefore, we made the frequency of its application a parameter of the genetic programming system (also suggested by Koza [7]).

4 Experimental Results

Experiments were conducted on two symbolic regression problems. The goal was to evolve the programs that approximate the functions (1) $F_1(x) = x + x^2 + x^3 + x^4$ and (2) $F_2(x) = 1.5 + 24.3x^2 - 15x^3 + 3.2x^4$, respectively, in 100 data points, that were randomly selected in the $[0, 1]$ interval. The parameter setting is shown in Table 2.

Table 2. The genetic programming parameter setting

Objective	Evolve a function that fits the data points of the fitness cases
Terminal set	x, real numbers $\in [-100, 100]$
Function set	$+, *, /$
Fitness cases	$N = 100$ randomly selected data points (x_i, y_i), (1) $y_i = x_i + x_i^2 + x_i^3 + x_i^4$ (2) $y_i = 1.5 + 24.3x_i^2 - 15x_i^3 + 3.2x_i^4$
Raw fitness and also standardized fitness	$\sqrt{\frac{1}{N}\sum_{i=1}^{N}(gp(x_i) - y_i)^2}$, $gp(x_i)$ being the output of the genetic program for input x_i
Population size	100
Crossover probability	90%
Probability of Point Mutation	10%
Selection method	Tournament selection, size 10
Termination criterion	none
Maximum number of generations	50
Maximum depth of tree after crossover	20
Initialization method	Grow
Frequency of simplification	Every 1., 2. or 5. generation
Simplification probability	0-100%

In the fitness measure only the error is included, there is no term related to the program size. We think that *a shorter program with more errors should not be preferred to a longer program containing less errors.* The mechanism for limiting code growth should be distinct from the selection mechanism. We introduced the simplification as a mutation operator in order to reduce code size without influencing the fitness-based selection mechanism.

Table 3. Parameter setting for simplification

Frequency F[gen.]	Probability P[%]
-	0
1	10, 20, 100
2	20, 40, 60
5	20, 25, 50, 75, 80, 100

We added two parameters: the frequency of simplification (F) and the simplification probability (P). Simplification is applied every F-th generation, on every individual program with probability P.

For each parameter setting shown in Table 3 we performed 50 runs and recorded their average. The plots for the regression of F_2 are presented, since it is a more difficult problem. Nonetheless, the plots for F_1 have the same character.

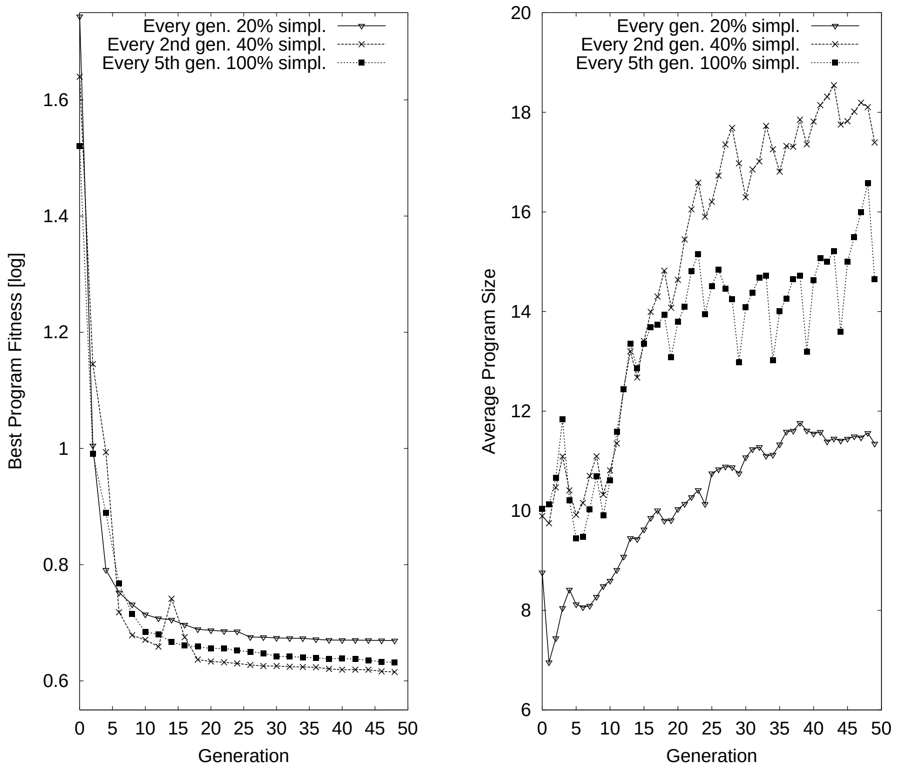

Fig. 1. The best program fitness and the average program size over generations, for $P/F = 10\%$

In order to establish a good ratio between the probability and the frequency of simplification, we compared the results of the runs with the same overall simplification ratio P/F. ($P/F = 10\%$ is achieved when (1) 10% probability of simplification in each generation, (2) 20% probability of simplification in every second generation or (3) 50% probability of simplification in every fifth generation is applied). Particularly, we analyzed the results for $P/F = 10\%$ and $P/F = 20\%$. The results for $P/F = 20\%$ are shown in Figure 1.

Considering program size, best results were obtained when simplification was applied in every generation: code growth practically stopped at the 35th generation. In the case of 100% simplification in every fifth generation, the form of the graphics (Figure 1 right) clearly reflects the alternating behavior of program size: after every fifth generation the average program size is reduced by simplification, then programs are allowed to grow for the next five generations. In fact, if we look at the average program size at every fifth generation (after simplification), we can see that in this case code growth is moderated after the 30th generation. In the meantime, the fitness of the best program (Figure 1 left) was slightly lower when simplification was applied less frequently (and with higher probability, keeping P/F constant).

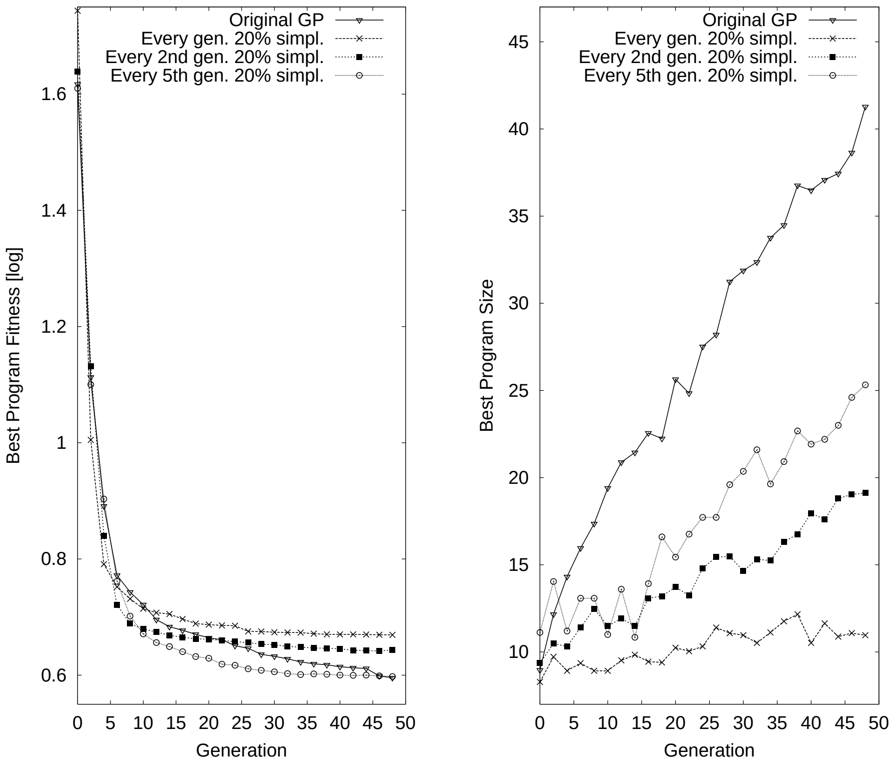

Fig. 2. The best program fitness and size over generations

We made another comparison among the results of runs with different frequencies of simplification, keeping the probability constant at 20%. Figures 2 and 3 show the results for the cases: (1) no simplification (original genetic programming), (2) simplification in every generation, (3) every second or (4) every fifth generation. The results for simplifying every individual in every generation ($F = 1$, $P = 100\%$) are very close to those obtained when simplifying in every generation

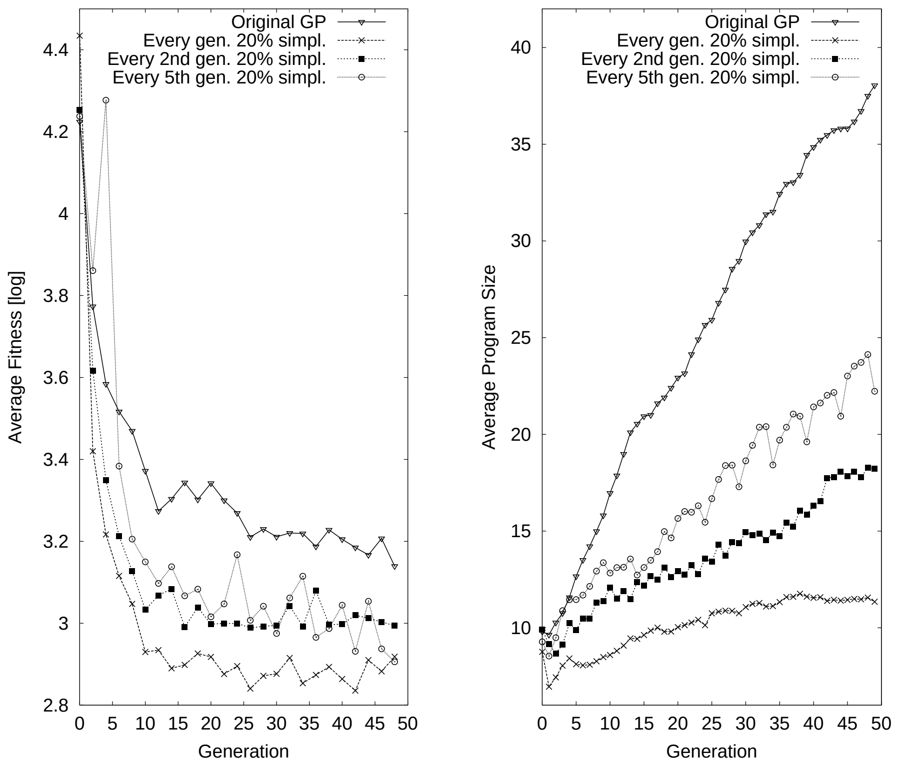

Fig. 3. The average program fitness and size over generations

with probability $P = 20\%$ (since the corresponding graphics are practically undistinguishable, we do not show the graphics corresponding to total simplification). As we expected, in the case of genetic programming without simplification, the program size is continuously growing (on the average, 0.5 $nodes/generation$) and in the case of simplification in every generation, the program size remains at a low value. In the latter case, by averaging over 50 independent experiments, the code growth is 0.08 $nodes/generation$ (for the average individual) for the first 28 generations, and 0.02 $nodes/generation$ for the last 22 generations. The fitness of the best program (Figure 2) is slightly worse when simplification is applied more often. The average fitness (Figure 3) of the cases when using simplification is better than that of the case with no simplification.

We also compared the results of runs, when the frequency of simplification was constant (every fifth generation) and the probability of simplification varied between 0-100%. While the size of programs was growing fast when no simplification was applied, it was quite stable when the probability of simplification was high. In the case of simplifying each individual program, the accuracy of the best program was slightly worse than for the other cases. Thus, we found again that applying simplification more often leads to much shorter, but slightly worse solutions.

Since a better performing program is a program with lower fitness and lower size, we could represent the performance of genetic programs as the product of fitness and size (Figure 4). In this view, the results are better when simplification is applied more often. But if we represent the performance as $Fitness^{N_F} * Size^{N_S}$, N_F and N_S being integers, if $N_F > N_S$ (meaning that fitness is more important than size), then the differences shown in Figure 4 diminish, or even disappear.

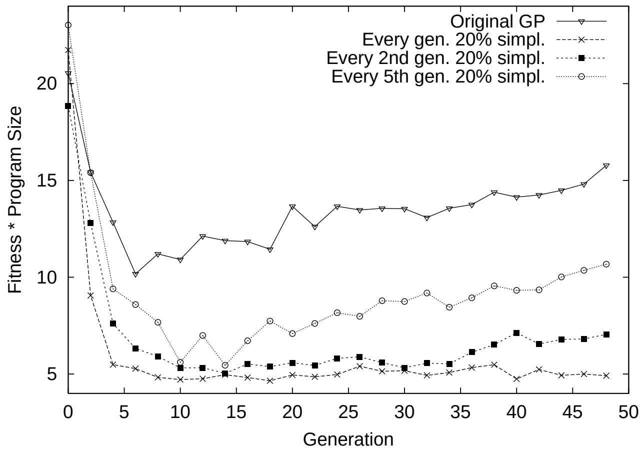

Fig. 4. Performance of the best fitness producing program

When choosing the probability of simplification, one has to make a trade-off between accuracy and program size:

– more accurate programs that grow moderately (less simplifications); or
– less accurate programs that do not grow (more simplifications).

5 Application

Our interest for limiting code growth in symbolic regression problems stems from the need for reducing the CPU time of our machine learning system [3]. The system is based on constructive induction, having the two components:

– learning engine - the C4.5 decision tree learning program [10]; and
– new feature generator - genetic programming.

We use this system in solving a mechanical engineering design problem, namely four bar mechanism synthesis [11]. The learning task is to discover the structural description of classes of such mechanisms. When generating the structural description, genetic programming creates the new features as algebraic functions of the six structural parameters of four bar mechanisms (simple symbolic regression with a terminal set consisting of six variables).

Genetic programming is called at the creation of each node in the decision tree (about 50 times in each run), on the data that has to be classified at that node . The training data set (i.e., the fitness cases) contains more than 7000 items. Thus, evaluating irrelevant code on these data takes much CPU time.

Initially, we tried to solve the problem by just using genetic programming, but the results were poor (there were too many errors in the produced description). The results produced by decision tree learning were better, but in many cases the original attributes could not describe correctly the classes. By allowing the creation of new attributes (algebraic expressions over the original attributes) at each node of the decision tree, the performance of the system improved considerably (from 47 rules and 6 misclassified cases when using C4.5 alone to 37 rules and 4 misclassified cases when using C4.5 with genetic programming at node level).

6 Conclusions

In the paper a method for limiting code growth in genetic programming was presented. The mechanism for controlling the size of programs was distinct from the fitness-based selection mechanism. The control of code growth was realized by a special mutation operator, inspired by three forms of mutation in biology. The method was applied to symbolic regression, where the special mutation operator consisted in algebraic simplification. The evolution of programs had two alternating phases:

– classical genetic programming - allowing code growth and intron formation (several generations); and
– simplification of programs - eliminating introns and reducing program size by means of a special mutation operator (one generation).

The experimental results show that code growth can be moderated or even stopped without the deterioration of performance by choosing the right frequency and probability for the application of simplification.

Acknowledgements

This work has been supported by grants No. T023305 and T25471 of the National Research Foundation of Hungary. The author is grateful to A. Márkus and J. Váncza for many helpful discussions.

References

1. Peter J. Angeline, 'Genetic programming and emergent intelligence', in *Advances in Genetic Programming*, ed., Kenneth E. Kinnear, 75–97, MIT Press, (1994).
2. Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone, *Genetic Programming: An Introduction*, Morgan Kaufmann, 1998.
3. Anikó Ekárt, 'Generating class descriptions of four bar linkages', in *Late Breaking Papers at the Genetic Programming 1998 Conference*, ed., John R. Koza, pp. 42–47, (1998).
4. Walter Gilbert, 'Genes-in-pieces revisited', *Science*, **228**, 823–824, (1985).
5. Dale C. Hooper and Nicholas S. Flann, 'Improving the accuracy and robustness of genetic programming through expression simplification', in *Genetic Programming 1996: Proceedings of the First Annual Conference*, eds., John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, p. 428, (1996).
6. Hitoshi Iba, Hugo de Garis, and Taisuke Sato, 'Genetic programming using a minimum description length principle', in *Advances in Genetic Programming*, ed., Kenneth E. Kinnear, 265–284, MIT Press, (1994).
7. John R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press, 1992.
8. William B. Langdon, 'Size fair and homologous tree genetic programming crossovers', in *GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference*, eds., W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, (1999).
9. Peter Nordin, Frank D. Francone, and Wolfgang Banzhaf, 'Explicitly defined introns and destructive crossover in genetic programming', in *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, ed., Justinian P. Rosca, pp. 6–22, (1995).
10. J. Ross Quinlan, *C4.5: Programs for Machine Learning*, Morgan Kaufmann, 1993.
11. George N. Sandor and Arthur G. Erdman, *Advanced mechanism design: Analysis and synthesis*, volume 2, Prentice Hall, 1984.
12. Terence Soule, James A. Foster, and John Dickinson, 'Code growth in genetic programming', in *Genetic Programming 1996: Proceedings of the First Annual Conference*, eds., John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, pp. 215–223, (1996).
13. Lubert Stryer, *Biochemistry*, Freeman, 1995.
14. J. Watson, N. H. Hopkins, J. W. Roberts, J. Argetsinger-Steitz, and A. M. Weiner, *Molecular Biology of the Gene*, Benjamin-Cummings, 1987.
15. Allan C. Wilson, 'The molecular basis of evolution', *Scientific American*, **253(4)**, 148–157, (1985).
16. Byoung-Tak Zhang and Heinz Mühlenbein, 'Balancing accuracy and parsimony in genetic programming', *Evolutionary Computation*, **3(1)**, 17–38, (1995).