Genetic Programming with Dynamically Regulated Parameters for Generating Program Code

Tomasz Łysek and Mariusz Boryczka

Institute of Computer Science, University of Silesia,
ul.Będzińska 39, Sosnowiec, Poland
{tomasz.lysek,mariusz.boryczka}@us.edu.pl

Abstract. Genetic Programming (GP) is one of the Evolutionary Algorithms. There are many theories concerning automatic code generation. In this article we present the latest research of using our dynamic scaling parameter in Genetic Programming to create a code. We have created practically functioning program code with the dynamic instruction set for *L* language. For testing we have chosen the best known problems. Our investigations of the best range of each parameter were based on our preliminary experiments.

Keywords: Genetic Programming, Linear Genetic Programming, Dynamic Parameters, Code generation

1 Introduction

Genetic Programming (GP), one of Evolutionary Algorithms has been developed mainly by John Koza and Wolfgang Banzhaf between 1992 and 2007 [4]. The greatest improvement of the algorithm has been suggested by Banzhaf and Bremaier approach to Linear Genetic Programming. They have proposed an proprietary solution that depends on tournament selection and strict machine code for individuals structure. Genetic Programming is an extension of Genetic Algorithm, and one of the population algorithms based on the genetic operations. The main difference between those two is the representation of the structure they manipulate and the meanings of the representation. Genetic Algorithms usually operate on a population of fixed-length binary strings, GP typically operates on a population of parse trees that usually represent computer programs [5]. There are various models of an individual structure in GP population as well as various methods to modify an individual - by crossing-over or mutation. It is common that higher value of the crossover probability will result in better exploitation and high mutation will improve the exploration. Based on our previous research we have created a possibility of dynamic regulation of the parameters, responsible for the probability of the certain manner of modify an individual, in a way that it will increase the speed of finding better results and earlier detect stagnation of population at the same time [2]. We created a dynamic control parameter

- responsible for the course of the algorithm that in fewer iterations receives better value of the fitness function, comparing to the results known from the literature. We have managed to reduce the length of a code of the individual as well. Our aim is to adapt the dynamic control parameter so that it will generate a program code which would be capable of solving programming issues depending on collection of input-output vectors. This article is organized as follows: first we analyse related works and ideas of creating most effective GP in the literature. Afterwards, we present the GP theory, that is the base to our research and experiments. The fourth section is dedicated to our idea of the dynamic control parameter and experiments proving its effectiveness. In the last section we describe the course of the experiment associated with a program code generating and we present its applicability. We compare our proposal with the results achieved by classical GP algorithms. We summarize with short conclusions.

2 Related Work

Genetic Programming was applied to various domains by Koza. Further development of this method involves modify of the population structure, of the type of an individual structure and introducing new methods of genetic operations [1]. Classical approach assumes representation of an individual by tree structure (Tree-based GP). There are also some modifications in which an individual is represented through Rule-based GP (Rule-based GP), which is gene expression for Genotype-Phenotype Mappings (GPM) by Ferreira [8], which assumes that individual's structure is a string with a head and a tail. The head is a list of expressions (functions and symbols) and the tail is a list of the arguments. Linear Genetic Programming (LGP) algorithm proposed by Koza and improved by Banzhaf and Bremaier turned out to be the breakthrough. LGP is based on presenting an individual in a graph structure, which vertices are the program instructions. Brameier introduced population divided into two groups and the leaders of those groups are crossed [7]. This is a huge leap from the classical approaches with tree-based code. For the purposes of the experiments classical approach TBGP and linear approach LGP has been tested. An individual modifications (mainly through different mutation types) has been taken from the literature and implemented according to given patterns. Experiments regarding effectiveness of modification and setting probability intervals of choosing control parameters has been conducted in 2012 and 2013.

3 Genetic Programming and Linear Genetic Programming

Genetic Programming described by John Koza is an algorithm processing test input vectors into their corresponding output vectors. Koza has defined as well five steps to be performed in order to solve a problem using GP:

1. Define the terminal set,
2. The function set,
3. Define fitness measure,
4. Select control parameters,
5. Define termination and result designation.

Individuals in GP are build with instructions made in defined L language. Collection of programs in L language is called genotype G. Phenotype P is defined as a set allowing reflection of input vector into output vector.

$$\begin{gathered} f_{gp} : I^n \to O^m : f_{gp} \in P \\ gp \in G \end{gathered}$$

The defined approximation target is to find the best T from the given collection using the evolution process:

$$T = \{(\boldsymbol{i}, \boldsymbol{o}) | \boldsymbol{i} \in I' \subseteq I^n, \boldsymbol{o} \in O' \subseteq O^m, f(\boldsymbol{i}) = \boldsymbol{o}\}$$

Evaluating of the fitness function is determined by detection of the error size made by each individual. In order to exacerbate selection requirements of the best individual, to the fitness function we add modifications as a penalty e.g. late iterations or a tree depth/length of the generated code. A popular way of determining errors in approximation tasks is the sum of squared errors (SSE) [3]. The mean square error for SSE is evaluated from equation:

$$MSE(gp) = \tfrac{1}{n} \textstyle\sum_{k=1}^{n} (gp(\boldsymbol{i_k}) - o_k)^2$$

LGP algorithm differs from the classical approach mainly in an individual construction, a population structure and the number of parameters improving its effectiveness. An individual in LGP algorithm is build in the way resembling a program code in the machine language. LGP structure is a combination of the idea of creating a genotype with the binary code and the idea of programming using genes. Genotype in the form of the binary code (RBGP - Rule-based genetic programming) presents coding the set of symbols (terminals) and operations by means of appropriate binary values. Each individual of the population consists of many classifiers processing I^n into O^m. In this algorithm a crossover and a mutation are performed in a classical way, whereas the fitness function is based exclusively on the high value of comparing an individual's genotype with encoded version $\boldsymbol{o} \in O' \subseteq O^m$. Individual's structure presented by genes connections (GPM Genotype-phenotype programming) has been presented by Ferreira. He based it on dividing genotype into two groups: a head and a tail. The head is a list of functions and operators used in an individual. The tail is a list of arguments provided program as a component of the L language. Additional symbols introduced in functional part resemble registers used in LGP algorithm. The algorithm of the linear Genetic Programming combines features of RBGP and GPM, because the structure of the individual based on the programming code contains encoded values in form of registrations. In some modifications, at the beginning of each individual appears a headline containing functions and symbols used in the certain genotype. Thanks to graph-like structure of an individual even at the very first tests conducted by Banzhaf it has been demonstrated that LGP is a better method for solving more elaborated problems. An ultimate advantage of LGP over the classical approach has been revealed in Bremaiers scientific work. In the TBGP there is only a limit of the tree depth. There are two basic parameters restrictive creating genotypes in LGP, these are maximal length of the code (number of code's lines) and maximal length of the single code line (determined on the base of the number of operators and symbols). Main features of LGP are:

- the structure of command list (that can be presented in the form of directed graph), instead of classical approach based on a tree,
- the linear structure of the program performed as a processor machine code,
- acquired values are saved in the dynamic registers which behaves like inner processor registers,
- subgraphs formed inside of an individual, used as functions, registers, are treated as variables,
- inner algorithm searching and deleting inefficient code through individual's evaluation, based on their fitness function,

Assuming that fulfilment of the four out of five point out of five Koza's steps is being represented by the collection of the parameters:

- probability of crossover differs two individuals by crossing-over chooses parts,
- probability of mutation changes part of an individual with by specified way,
- maximum numer of individuals in population restricts the number of individuals,
- maximum tree depth/maximum code lines restrict individual size,
- probability of changing function/terminal chenges function or terminal to other from L in tree node/line of code,
- probability of permutation swap over pieces of the tree (TBGP),
- probability of inserting / deleting adding randomly generated part from L to individual/deleting random part of individual,
- probability of encapsulation protection part of individual against further changes,
- probability of automated defined function (ADF) - Recognition of useful fragments of genotype and transfer the parts to the set of available features.

In the classical TBGP, as well as in each modification there is a number of parameters influencing the quality of obtained results. Various parameters and their values can be used depending on the studying problem. Most of the parameters are flexible and can be used in the TBGP algorithm as well as in the LGP. However there are prepared special parameters, that can only be used in the particular versions of genetic programming. For the GP based on tree structure there is a possibility of the mutation through lifting a fragment of the tree for selected number of levels, and putting it in other node's place, while in LGP there is a possibility of limiting the number of the registers. Those solutions cause the loss of some parts of the genotype, shortening its length. Combining that with encapsulation or automatic defined functions leads to improving the results.

4 Dynamic Parameters in Genetic Programming

In Luke's and Spector's experiments it has been demonstrated that simple mutation and standard crossover in case of genetic programming algorithms affects results to the same degree [5]. However the crossover works well for large populations, while the mutation allows to obtain better outcomes for smaller populations with a larger number of iterations of the algorithm. When the crossover and the mutation is used the most important problem is selecting a node. A mutation needs to have specified a node to which it is applied, and a crossover needs to have chosen two nodes, from which it starts the operation. In the literature, the most frequently discussed is an example of a random selection. The research was used Weise method, where the base is factor defining the weight of a subtree of the test fragment [8]. Weise weighting factor is based on the assumption that the best selection will be selecting all nodes c and n tree t, with the same probability distribution as in the case of random select, eg. $P(nodeSelection(t) = c) = P(nodeSelection(t) = n) \forall s, n \in t$. Weight node n is obtained by the number of nodes in the subtree of n:

$$W(n) = 1 + \sum_{i=0}^{l(succ(n))-1} W(succ(n)_i),$$

where W is function that determines weight of node n, $succ(n)$ is function that determines set of child nodes of n, and l is function determines the lenght of the n subtree.

Algorithm 1: Setting node weight

```
begin
1   $flag = true$; $c = t$;
2   while $flag$ do
3       $r = \lfloor random(0, W(c)) \rfloor$;
4       if $r \geq W(c) - 1$ then
5           $b = false$;
        else
6           $i = l(succ(c)) - 1$;
7           while $i >= 0$ do
8               $r = r - W(succ(c)_i)$;
9               if $r < 0$ then
10                  $c = succ(c)_i$; $i = -1$;
                else
11                  $i = i - 1$;
12  return c;
```

The evaluation function has a major impact on the structure of the population in subsequent iterations. We propose the construction of the evaluation function based on the MSE and the additional penalty rates for a large number of iterations of the algorithm and the length of the code of individual:

$$\mathrm{FO}(gp) = \mathrm{MSE}(gp) + \mathrm{KI} + \mathrm{KW}(gp)$$

where:

- KI designated punishment for a long iteration
- KW(gp) designated penalty for a large depth of the tree (TBGP) / large number of lines of code (LGP)

Dynamic parameters involves adding to the algorithm scaling factor based on the results of studies on the extent to which the crossover and mutation parameters allows to get the best possible result. In addition, studies have been carried out concerning the designation of a minimum number of iterations that achieve the high value of fitness function [2]. In table 1 and 2 we present test results (F function value tending to 0; G depth of generated tree for TBGP; D length of generated program for LGP).

Table 1. Setting the parameters for the best intervals TBGP and LGP (part 1)

	function change				terminal change				permutation				inserting			
	TBGP		LGP		TBGP		LGP		TBGP		LGP		TBGP		LGP	
Value	F	G	F	D	F	G	F	D	F	G	F	D	F	G	F	D
0.1	13.9	19	4.7	192	14.9	20	4.3	200	15.2	20	6.2	200	**14.5**	**17**	**4.7**	**189**
0.3	**13.6**	**17**	**3.7**	**185**	13.8	17	**3.8**	**186**	**14.5**	**18**	**4.9**	**186**	14.9	19	5.1	194
0.5	**13.5**	**17**	**3.7**	**191**	**13.7**	**18**	**3.7**	**194**	**14.3**	**19**	**5.1**	**194**	15.2	20	5.3	198
0.7	14.1	20	5.4	200	14.1	20	4.9	200	14.9	20	5.7	200	15.7	20	6.4	200

Table 2. Setting the parameters for the best intervals TBGP and LGP (part 2)

	cutting				encapsulation				ADF				lifting	
	TBGP		LGP		TBGP		LGP		TBGP		LGP		TBGP	
Value	F	G	F	D	F	G	F	D	F	G	F	D	F	G
0.1	**14.9**	**16**	**4.2**	**186**	**14.1**	**16**	**4.2**	**188**	14.2	18	**4.7**	**193**	13.8	18
0.3	**15.1**	**18**	**4.9**	**189**	14.4	17	4.6	193	**13.8**	**14**	**4.2**	**186**	**13.6**	**16**
0.5	15.6	18	**5.3**	**194**	15.3	19	4.9	197	**14**	**14**	5.4	182	13.9	17
0.7	15.9	19	5.4	197	15.8	20	5.1	199	14.5	14.5	9.6	179	14.3	20

On the basis of these studies was created scaling parameter γ, that value at the beginning of solving the problem is 1, and in subsequent iterations oscillates between the values that $\langle 0, 1\rangle$ [2]. If the parameter is set to a value of 1 or close to, then occurs the exploitation of searched space solution and the value of the fitness function of the population will converge. If the value of the parameter γ would be closer 0, then the population will be a subject of a greater number of mutations and there will be a greater exploration of the solution space. Fitness function value of individuals in successive iterations of the algorithm may get closer to the results from the initial sampling algorithm or possess a better result.

In such a case, the parameter γ is changed to the exploratory character not to allow an excessive convergence, and acquired results can be add to the collection of output vectors. In case of a deterioration the γ parameter is transformed into the exploitation form.

$$\mathrm{PM}(M_i) = \begin{cases} \frac{\mathrm{SW}(M_i)}{\gamma} \text{ if encapsulation or inserting} \\ \\ \mathrm{SW}(M_i) \cdot \gamma \text{ , for other cases} \end{cases}$$

where:

- M is collection of the possible mutations,
- PM is the function giving a new value of probability for drawing the M mutation,
- SW is a weighted average based on values of the fitness function FO, upper and lower edges of the best range of the values of probability for mutation M_i.

Algorithm 2: The algorithm for determining the value of a new mutation

```
1 Determine the degree of the population stagnation
2 Establish a new $\gamma$ value in $(0, 1\rangle$ (progressing stagnation of the population
  involves $\gamma$ parameter is getting closer to 0),
3 Designate a weighted average of the lower and upper edge of the best range of
  values for a set of mutation probabilities
4 Fixing a new probability values for each mutation
```

Example:

- The m stagnation appears,
- A new γ value is calculated $\gamma = \begin{cases} \gamma - 0.1 \text{ , } \gamma > 0.1 \\ 0.1 \text{ , } \gamma = 0.1 \end{cases}$,
- Cutting mutation value according to table 2 $SW = 0.2$ must be replaced by $PM = SW \cdot \gamma$ in order to achieve new probability.
- For each M_i element it is necessary to determine a new PM.

5 Experiments and Results

The purpose of experiments was to compare effectiveness of GP and LGP in C++ to the same algorithms written and compiled in authors' platform [6]. To make our algorithm more efficient we added smart code completion that checks if used function in generated individual needs to add functions library from programming language. This mechanism will provide smaller start collection of terminals in language L and in further point of iteration will decrease consumption of adding new libraries. Classical GP-like algorithm with proposed modification and smart code mechanism is as follow:

Algorithm 3: Modified Genetic Programming algorithm

```
1  Generate population P with random composition of defined functions;
2  while stop criterion is not met do
3      Parse generated individuals (programs) to set value of fitness function;
4      Copy the best individual;
5      Calculate the weight of the obtained results to determine the degree of
       convergence of the population;
6      Change the value of $\gamma$;
7      Calculate the Weise weight for selected fragments of individuals;
8      Create new programs using mutation and crossover;
9      Check the length of the algorithm;
10     Check the length function;
11     Check the depth of nesting;
12     Append missing libraries;
13 An individual whose genotype achieved the best result of the adaptation
   function can be exact or approximate solution.
```

Test problems:

– Loop Input vector: value that determine loop stop, value that determine loop step. Output: collection of values generated by loop,
– Factorial - Input vector: value for factorial. Output: factorial for value,
– Fibonacci - Input vector: value of Fibonacci number. Output: value of Fibonacci number,
– GCD - Input vector: number 1, number 2. Output: GCD for two numbers,
– Bubble sort - Input vector: collection on values. Output: sorted collection of numbers,
– Distinct roots Input vector: values of delta, b and a. Output: distinct roots values.

Parameters for experiments were:

– the size of population $N = 500$,
– the crossover parameter $CR = 0.9$,
– the mutation parameter $F = 0.1$,
– the maximum number of iterations is equal to 500,
– the maximum tree depth (TBGP) is equal to 20,
– the maximum operator nodes is equal to 200,
– the maximum program length (LGP) is equal to 200,
– for every testable function the algorithm was run 10 times,
– the maximum algorithm length: $algLen = 100$,
– the maximum function length: $funLen = 10$,
– the maximum depth of nesting: $maxDepth = 2$,
– percent of the population of test subjects: $testBoids = 5$.

Table 3. Medium percentage errors of best individuals in the population (part 1)

	Loop				Factorial				Fibonacci			
	TGP	TGP (γ)	LGP	LGP (γ)	TGP	TGP (γ)	LGP	LGP (γ)	TGP	TGP (γ)	LGP	LGP (γ)
50	28.3	29.9	21	22.1	57.9	59.1	55.9	56.8	70.3	76.1	66.1	66.4
100	28	28.1	20.3	19.2	56.4	58.4	54.1	54.2	65.1	67.9	65	64.7
150	27.8	27.9	19.7	17.6	53.1	55.9	49.8	50.7	61.8	63.8	59.4	59.3
200	27.4	25.3	19.4	**16.1**	49.1	49.3	46.2	46.6	56.2	55.9	53.1	53.7
250	26.5	24.1	18.9	**14.9**	43.9	43.4	41.6	39.3	49.6	49.5	47.6	46.2
300	25.2	**22.3**	18.2	**14.1**	41	39.2	38.4	33.9	47.3	45.1	44.4	40.9
350	24.3	**21.6**	17.8	**13.5**	38.4	32.8	33.9	29.1	45	**40**	39.6	37.1
400	23.6	**21.4**	17.1	**13.2**	32.6	**28.4**	29.1	**25**	42.9	**37.4**	37.5	**33.8**
450	23.1	**21.4**	16.8	**12.9**	30.2	**26.1**	27.8	**23.4**	41.2	**36.1**	35.1	**31.4**
500	22.8	**21.4**	16.7	**12.9**	29.4	**25.9**	26.1	**22.5**	40.3	**35.7**	34.9	**30.2**

Table 4. Medium percentage errors of the best individuals in the population (part 2)

	GDC				Bubble sort				Distinct roots			
	TGP	TGP (γ)	LGP	LGP (γ)	TGP	TGP (γ)	LGP	LGP (γ)	TGP	TGP (γ)	LGP	LGP (γ)
50	91.6	91.4	84.7	88.2	96.4	98.6	83.7	88.9	90.2	93.9	89.4	91.6
100	90.9	89.6	79.1	81	84.5	85.7	79.9	81.3	85.2	89.4	83.1	85
150	89.1	81.2	73.5	75.9	82.6	82.1	76.6	75.4	80.1	80.6	73.9	74.2
200	84.9	74.7	67.9	68.1	80.1	76.4	72.9	70.2	76.7	78.1	64.8	61.9
250	70.6	61.9	55.6	55.3	76.5	72.2	68.5	62.3	72.8	72.5	59	56.4
300	62.7	51.8	49.4	46.8	69.8	68.5	62	58.6	66.2	64.9	56.7	52.8
350	53.8	45.5	45.3	42.3	65.2	63.8	58.3	53.8	61.9	58.7	50.3	48.9
400	49.2	**39.4**	42.5	**38.5**	61.4	60.2	56.7	52.1	54.3	50.1	48.7	46.7
450	46.3	**37.9**	40	**36.2**	59.7	58.9	52.1	**51.3**	49.8	**44.6**	45.1	**42.5**
500	44	**36.1**	39.1	**34.9**	58.4	**58.1**	51.6	**50.9**	48.3	**42.1**	44.5	**39.8**

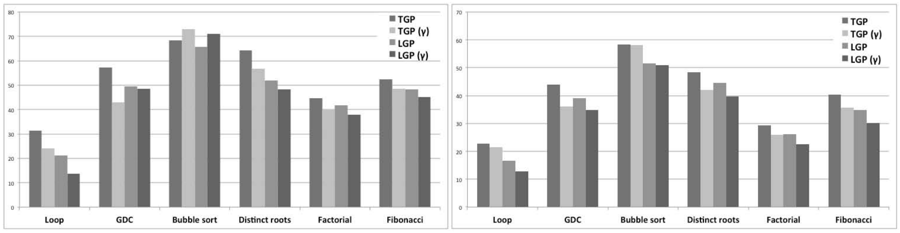

Fig. 1. Median error (left) and minimum error (right) in individual

Tables 3 and 4 show the results of the best individuals in a given iteration of the algorithm. Adding a γ parameter allowed improving the results obtained in each test. In addition, our algorithm acquires a better result long before the classical approach. As expected from previous studies, when more than 500 iterations stagnation of the results appears[2]. In fig. 1 we presented that the median results of the algorithm with use of γ parameter does better than the classical approach (with the exception of bubble sort algorithm). In fig. 1 you can also notice that the minimal error of the best individual is significantly reduced comparing to the classical GP algorithms. In the box plot it an be noticed that in most cases the population is concentrated around better solutions, and individuals with large error are marginalized. Furthermore we have noticed a strong

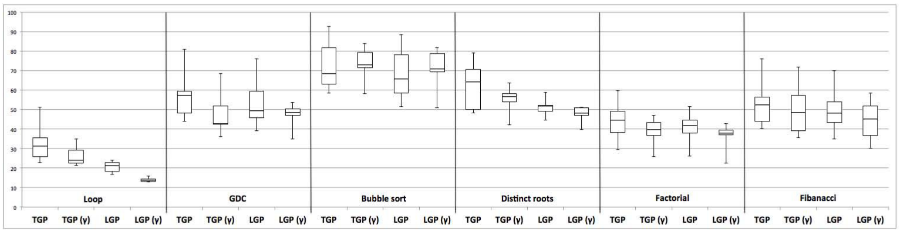

Fig. 2. Box plot of iteration 500

tendency of the population with good solutions to concentrate while maintaining the margin to the possibility of exploration.

6 Conclusions and Future Work

We have provided a better algorithm for generating a program code. By applying the proposed modification created program code is usable as a template code or the initial solution generator. Proposed γ parameter allows significant acceleration for better performance and reduces the size of individuals. The proposed algorithm in accordance with earlier experiments confirms the validity of the application of the populations of not less than 300, but not more than 500 because of constant stagnation results. LGP algorithm is definitely better adapted to the problem of the program code generation, mostly due to the structure of the individual. Future research will be based on checking whether the proposed genetic programming algorithm can be adapted to solve the hashing problem.

References

1. Banzhaf, W., Nordin, P., Keller, R., Francone, F.: Genetic Programming - An Introduction, pp. 133–134. Morgan Kaufmann Publishers (1998)
2. Łysek, T., Boryczka, M.: Dynamic parameters in GP and LGP. In: Nguyen, N.T., Trawiński, B., Katarzyniak, R., Jo, G.-S. (eds.) Adv. Methods for Comput. Collective Intelligence. SCI, vol. 457, pp. 219–228. Springer, Heidelberg (2013)
3. Brameier, M., Banzhaf, W.: Linear Genetic Programming, pp. 130, 183–185, 186. Springer (2007)
4. Koza, J.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press (1992)
5. Luke, S., Spector, L.: A Comparison of Crossover and Mutation in Genetic Programming (1997)
6. Łysek T.: Dedicated language and MVC platform for Genetic Programming algorithms. Journal of Information, Control and Managament Systems (2012)
7. Nedjah, N., Abraham, A., de Macedo Mourelle, L.: Genetic Systems Programming: Theory and Experiences, pp. 16–17. Springer-Verlag (2006)
8. Weise, T.: Global Optimization Algorithms: Theory and Application, pp. 169–174, 191–195, 207–208 (2009)