DOI: 10.17587/it.26.3-8

S. Khashin, PhD, Professor, e-mail: khash2@gmail.com,
S. Vaganov, PhD Student, e-mail: pro100-pioner@mail.ru,
Ivanovo State University, Ivanovo, 153025, Russian Federation,

Corresponding author: **Khashin Sergei,** Ivanovo State University, Ivanovo, 153035, Russian Federation, e-mail: khash2@gmail.com

Genetic Algorithms Using Forth

A method for automatic finding of a program (in the FORT language) that realizes the given algorithm is developed. The algorithm is specified by a set of tests of the form (input_data) → (output_data). Both input and output data are represented as a sets of 4-byte integers. Genetic methods have made it possible to find the implementation of even relatively complex algorithms: decimal and binary digits of numbers, GCD, LCL, factorial, simple divisors, binomial coefficients, sorting of short sequences, highs, lows, calculation of polynomial values and others. The genetic approach allows you to build a program from separate blocks, "genes", which turned out to be suitable for at least some part of the test elements. Genetic methods made it possible to find the implementation of even relatively complex algorithms: decimal and binary digits of a number,, NOC, factorial, simple divisors, binomial coefficients, sorting of short sequences, maxima, minima, calculation of polynomial values and others. Our method starts by randomly sorting through short programs, extracting blocks ("genes") from them at least slightly suitable for the problem being solved. And then he builds a program using the found "genes". The set of used "genes" in the process of the algorithm is constantly being adjusted, improved. The complexity of direct enumeration grows exponentially with increasing program length. The genetic method we propose allows us in many cases to drastically reduce the volume of search. The FORT language is chosen because of its compactness: all listed algorithms are placed in no more than 10—15 commands. Although, if we take into account the genes found, the total length of the program will be significantly longer. In addition, the mechanism of embedding "genes" already, in fact, is in the language. Our method is configured to work with integers, but it can be applied to data containing real numbers, strings, etc. In the case of working with real numbers, the method can be considered as an alternative to the methods used in neural networks.

Keywords: *Genetic algorithm, Linear genetic programming, Evolutionary programming, Machine learning, Forth*

УДК 004.023 DOI: 10.17587/it.26.3-8

С. И. Хашин, PhD, канд. физ.-мат. наук, доц., e-mail: khash2@gmail.com,
С. Е. Ваганов, аспирант, e-mail: pro100-pioner@mail.ru,
Ивановский государственный университет, г. Иваново

Genetic Algorithms Using Forth

Разработан метод автоматического нахождения программы (на языке ФОРТ), реализующей данный алгоритм. Алгоритм задается в виде набора тестов (входные данные) → (выходные данные). И входные, и выходные данные представлены в виде наборов целых 4-байтовых чисел.

Генетический подход позволяет строить программу из отдельных блоков, "генов", которые оказались подходящими хотя бы для некоторой части тестовых элементов. Генетические методы позволили найти реализацию даже сравнительно сложных алгоритмов: десятичные и двоичные цифры числа, НОД, НОК, факториал, простые делители, биномиальные коэффициенты, сортировка коротких последовательностей, максимумы, минимумы, вычисление значений полиномов и другие.

Наш метод начинает работу со случайного перебора коротких программ, выделяет из них блоки ("гены") хоть немного подходящие для решаемой задачи. А затем строит программу с использованием найденных "генов".

Комплект используемых "генов" в процессе работы алгоритма постоянно корректируется, улучшается.

Сложность прямого перебора растет экспоненциально с ростом длины программы. Предлагаемый нами генетический метод позволяет во многих случаях радикально сократить объем перебора.

Язык ФОРТ выбран ввиду его компактности: все перечисленные алгоритмы помещаются в не более, чем 10—15 команд. Хотя, если учитывать найденные гены, общая длина программы будет существенно больше. Кроме того, механизм встраивания "генов" уже фактически есть в языке.

Наш метод настроен на работу с целыми числами, однако его можно применить и к данным, содержащим действительные числа, строки и т. д. В случае работы с действительными числами метод можно рассматривать как альтернативу методам, применяемым в нейронных сетях.

Ключевые слова: *генетический алгоритм, линейное генетическое программирование, эволюционное программирование, машинное обучение, Forth*

Inroduction

Our goal is to automatically obtain a program in some programming language that implements an algorithm, defined by a set of test's elements. To create such a program, we will use methods that in some sense have analogies in biology, in genetics. This approach is called "Genetic programming" (GP) [2, 6]. In all known applications, the desired algorithm is defined by a set of tests, for example, for the sum of two squares we would have:

$$1,\ 1 \to 2$$
$$2,\ -1 \to 5$$
$$0,\ 4 \to 16$$
$$\dots$$

In the early days of GP, the goal of GP was exactly the construction of a program that implements a given algorithm in the selected programming language. In this paper we return to this original goal and show that it can work effectively. Moreover, in the classical textbooks [7, 8], it was emphasized that the particular choice of a programming language is not of importance, as all languages are equivalent in power to the Turing machine.

In practice, this was not entirely true. The number of possible programs turned out to be too large and was highly dependant on the choice of the programming language. So generally speaking, the application of GP for searching programs did not give the results that they had hoped for at the beginning.

Significant achievements were obtained in the case of tree-based genetic programming, that is, when the program is presented in the form of a tree of calculations. However, in this case the obtained algorithms are much more complicated than we wish them to be. The classical example which is frequently used in textbooks and articles considers a program that implements computation $x \to x^2 + x + 1$ or even $x \to x^6 + x^5 + x^4 + x^3 + x^2 + x + 1$ [9, 17, 20, 25]. These GP obtained programs have no cycles. In this direction, there is an interesting work [18]. Authors managed to find programs that implement the following functions:

- nguen1: $x \to x^3 + x^2 + x + 1$,
- nguen2: $x \to x^4 + x^3 + x^2 + x + 1$,
- nguen3: $x \to x^5 + x^4 + x^3 + x^2 + x + 1$,
- nguen4: $x \to x^6 + x^5 + x^4 + x^3 + x^2 + x + 1$,
- keijzer4: $x \to x^3e^{-x}\cos(x)(\sin^2(x)\cos(x) - 1)$,

and several other functions of this kind.

Genetic methods yielded great results in approximating real-valued functions. Here GP methods can compete with neural networks and even interlock with them. For example, in [26] typical 10 regression problems for neural networks are solved using GP methods.

In [12, 15], the concept of Recurrent Cartesian Genetic Programming (RCGP) is introduced, in which the calculation graph can contain cycles. As a result, the authors managed to implement the functions $n \to n(2n - 1)$, $n \to 1 + n(n + 1)/2$, $n \to n(n^2 + 1)/2$, and finally, even the Fibonacci numbers. However, this is not a real replacement for full-cycle operators.

In [13], 29 tasks were proposed for testing genetic programming research. Most of them require working with real numbers, strings, arrays, with displaying information on the screen. Some of these tasks can be also handled by our approach if we leave the result in the stack instead of its output onto the screen.

The approach closest to ours is Linear Genetic Programming (LGP) method [6, 11, 17, 20, 23, 24]. In this method, the program is represented as a sequence of instructions from imperative programming language. Their typical set consists of 4 arithmetic operations (two arguments) and a number of mathematical functions, for example exp, sin, sqrt, etc. Practically all considered systems receive algorithms without cycles.

In [14] C++ is used. However, here the primary goal is the (minor) correction of already created (by students) programs.

In [10] describes automatic creation of the simplest program in the language Haskell. Again there are no cycles in the examples, even though there are calls to functions that work with arrays as a whole.

In [21] several GP methods are benchmarked on the 29 tasks from [13].

In the book [16] methods of GP in Python are considered. Here, more than a dozen different problems are solved by GP methods. The main difficulty in them is not the genetic algorithms themselves, but the suitable reformulation of the problem. After that, the problem can be solved by the genetic method, or directly.

The widely distributed machine learning package "TensorFlow" [3] also contains some tools for genetic programming. See more on this in, e.g., [19].

In fact, in almost all cases, the considered genetic methods are reduced to finding the minimum of certain loss function from some, possibly, quite a considerable number of parameters.

All these approaches takes us far away from the original idea of GP. In the this paper we instead stay within the framework of the originally formulated task. Algorithms are assumed to be quite complex, including conditional operators and cycles. Obviously, for a linear representation of the program one needs some version of the bytecode. Another problem is the method of accessing variables. In our opinion, one of the most compact way is the data stack. Programming language Forth language satisfies these conditions. Functions in Forth take all the arguments from the data stack and also leave the results of calculations there. In this paper, we restrict ourselves to the minimum possible set of tools: we don't use real numbers, strings, only 4-byte integers, we do not use variables, arrays, only data in the stack. Thus, we reduce Forth to a minimal subset of tools (only 32-bit integers, only stack, no registers, no variables), which allow us to solve quite complicated tasks, like GCD, binomial coefficient, primality testing, and so on. In the future, we would like to introduce other data types: real numbers, strings, etc. In addition, from the rather exotic language Forth, we would like to switch to some more common language (C, Java, etc.) or to an assembly language.

The key idea is to use partial programs, that is, programs that do not pass all tests, but only a part of them. We assume that fragments of such programs will be contained in the general program. We will use these fragments as "genes".

Overfitting. When training neural networks, the problem of overfitting often arises. In our case, this means that the program will work correctly only on the input data that are available in the test. This means that the program implements a tabular algorithm. This problem is completely eliminated in the present work as we introduce a limit on the length of the program. It is sufficient to require that the length of the program does not exceed the number of test elements.

Structure of the work. In section 1, we define the concept of a test and a test element. Section 2 contains a brief description of the version of Forth that we use. In section 3 we describe the methods that are used to find the Forth-realization, section 4 describes our results.

1. Problem definition

We want to find a program in some programming language implementing given algorithm. The algorithm will be defined as a set of tests:

$$(input\_data) \to (output\_data).$$

The unit of the test is the pair $t = (x, y)$, where $(x \in X, y \in Y)$, and X, Y are the corresponding sets of input and output objects. In this paper we consider as input and output objects as a sequences of 32-bit integers of fixes length.

Test files. A "test" is an arbitrary finite set of test items. Sometimes, when there are no confusion, we call a single test element a test. In practice, the test is represented by a text file of the form:

```
#T SHIFTR _ 0 x,y -> x*2^y comment
<in> 9 3 </in><out> 1536 </out>
<in> 4 3 </in><out> 48 </out>
<in> 2 7 </in><out> 28 </out>
...
```

Definition 1. Pair (m, n), where m is a number of input integers and n is a number of output integers is called the signature of the test element.

Remark 1. In general, it is allowed to have test elements with different signatures in one test. But this possibility is not used in the present work.

The number of test items in each test varies from a several to hundreds.

A large set of test files is freely available on our website [1]. Not for all of them our GP method was able to find implementing programs. The tests on the website are grouped by sections: polynomials, max/min, sorting, GCD, LCM, factorials, number theory, decimal and binary digits and so on.

2. Programming language

From our point of view all programs are divided into two types: linear (with goto) and structural.

The advantages and disadvantages of each type are quite obvious and there is no fundamental difference between them from our point of view. We have chosen a linear approach in our work (at least for this initial stage). Usually, this is done by selecting an assembly-type instruction set (registers). In the present work the stack approach (already implemented in the language of Forth [4, 5]) is deemed to be more effective. To find the program with the genetic method, the most stripped-down version of the Forth was chosen. Only those instructions are left, which are impossible to do without. There is no interaction with the user, such as output to the screen, and even variables are missing. The control structures are represented only by unconditional and conditional jump-instruction. Forth is very compact, new words (functions) are introduced very easily. The newly defined functions have exactly the same syntax as the built-in language elements. This is convenient for a genetic approach. Thanks to this, one does not need to go through programs with complex structure in the form of a tree, only the simplest ones.

An example of a program that calculates the sum of the squares of two numbers and the factorial:

```
   : SUMSQ2 DUP * SWAP DUP * +;
   : FACTORIAL CONST 1 OVER -- -ROT *
OVER IF -6 SWAP DROP;
```

All programs and functions in Forth work with the data stack. Only 4-byte integers are stored on the stack. The functions have no explicit arguments. The input data that they take is from the stack and they leave the results of their work in the same place.

The number of such implicit arguments can vary, depending on the state of the stack. Such a signature will be called "floating". For now, we will only consider functions with a fixed, static signature. At each point of such a program, the current stack depth is statically determined.

Everywhere below, functions will be called "words", or "genes". At each moment, the system has a certain set of built-in words (functions) and the current set of new genes, that is, new words built in the learning process ("evolution"). One step of evolution is the construction of one or more new words (functions) that solve one problem (test) in whole or in part.

Bytecode. The program in Forth is a byte sequence, each byte is a separate command. There are two types of commands: built-in and implemented on Forth itself. Thus, the total number of commands cannot be more than 255.

In our version of Forth there are 33 built-in commands: unconditional (GOTO) and conditional (IF) jump, numeric literal (CONST), stack manipulation commands (DUP DROP SWAP OVER ROT — ROT 2PICK 3PICK 4PICK 3ROLL 4ROLL), arithmetic (NEGATE +− * / % /% ++ − −), bit commands (AND OR XOR NOT), logical (comparison) (> < = 0 = 0> 0>).

In addition to the built-in commands, the system may contain commands implemented on Forth itself, for example, finding the sum of squares of two elements at the stack:

```
   : SUMSQ2        DUP * SWAP DUP * +;
```

The colon at the beginning of the command is a sign of the beginning of a new word. The name of the function follows. The semicolon at the end is a sign of the end of the word (function, program). The name of a function can be an arbitrary string of characters that does not contain spaces, tabs to the end of the line. Individual words are separated by spaces or tabs or line ends. An important advantage of Forth is that it is not required to develop a mechanism for "embedding genes", as it is already exist in the language. New words (procedures, functions, "genes") are used in exactly the same way as built-in ones. The current list of words (the dictionary) under biological interpretation will be considered as a "genome".

3. General algorithm

At first, let's just try to iterate through all the programs to a given length.

Working time. Having 33 basic words, we get $\approx 10^9$ programs of length 6.

If we check about 10 million programs per second (one core, frequency ≈3 GHz), then a full search will take about a minute. In an hour, you can go through all the programs of length 7. The search of all programs of length 8 requires more than a day. For an 8-core processor, it is realistic to check programs of length 9. Finding longer programs requires more complex methods.

A base step with parameters (L_0, L_1, T) in a fixed dictionary is the following algorithm.

1. A full search of programs up to length $\leqslant L_0$. Then build a list of partial programs (that is, programs that do not perform all tests, but only some of them).

2. Using the partial programs, build a list of word frequencies and a list of frequencies of word pairs.

3. Check random programs of length $L_0 + 1 ... L_1$ within a specified interval of time T seconds.

4. Correct the frequency tables and repeat step 3, this cycle is performed K (usually 8) times.

The typical values of the parameters for working on one processor core is (7, 14, 400), that is, first perform a full search of programs of length $\leqslant 7$, then 8 times for 50 seconds is performed alternately probabilistic/Markov search of programs of length from 8 to 14 with correction of the frequency table after each cycle.

Genes. As already mentioned, the list of words in the dictionary plays the role of "genome". Each gene added can be either "good", "useful", or "unsuccessful", "useless". The quality of the function-gene itself cannot be determined, only as part of the genome, aimed at solving a specific problem. Therefore, we introduce a definition of genome (dictionary) quality in relation to this set of tests. If the new gene improves the quality, we will consider it a good one. As the "candidates for genes" we will take the most frequent chains of bytes of length 2 or 3 among partial programs. During the process we will have two lists of programs (genes): "candidates for genes", and "unsuccessful genes". There should be no duplicate elements in them.

1. Find a list of valid words. Upon completion of the search, we get, besides this list, also a list of partial programs.
2. Clear the list "candidates for the genes" and "bad genes".
3. In the lists of "candidates for genes" add the most frequent chains of length 2, 3, and only those that are not in the list of "unsuccessful genes".
4. If there are no candidates for genes, we finish the job.
5. Add one of the candidates to the dictionary.
6. Perform a basic step with this genome, find its quality.
7. If the gene was unsuccessful, remove it from the dictionary and add it to the list of "unsuccessful genes".
8. Goto step 3.

4. Results

Here is a list of algorithms that have been implemented.

By brute force and probabilistic method. Squaring, odd, even, abs, sign, sum of two squares, max/min of two or three numbers, sorting of two numbers, GCD in special case ($x,y \geqslant 1$), factorial ($x \geqslant 1$), left/right shift in $k>0$ bits, minimal natural divider in special case ($x>1$). Decimal/hex/binary digits: lower, high.

Squaring:: `SQUARE DUP *;`
Multiply by 2: `MUL2 DUP +;`
Test odd: `ODD CONST 1 AND;`
Signum: `SIGN DUP 0 = IF 6 0> DUP IF 2 --;`
Sum of suares: `SUMSQ2 DUP * SWAP DUP * +;`
Absolute value: `ABS DUP 0> IF 2 NEG;`
Descending sort of 2 numbers:
`SORT2R OVER OVER > IF 2 SWAP;`
Ascending sort of 2 numbers:
`SORT2 DUP 2PICK > IF 2 SWAP;`
Maximum of 2 numbers:
`MAX2 OVER OVER > IF 2 SWAP DROP;`
Minimum of 2 numbers:
`MIN2 DUP 0> IF 2 SWAP IF 2 ROT;`
Maximum of 3 numbers:
`MAX3 OVER 3ROLL > IF 2 SWAP DROP;`
Minimum of 3 numbers:
`MIN3 ROT 2PICK 0> IF -4 DROP DROP;`
Polynomials:
$(a,b,x) \to bx + a$: `poly1 * +;`
$(a,b,c,x) \to cx^2 + bx + a$:
`poly2 -ROT 2PICK * + * +;`
$x \to 2x-3$: `pol1_1 -- DUP + --;`
$x \to -3x + 4$: `pol1_2 -- CONST -3 * + +;`
$x \to x-1$: `pol1_3 --;`
$x \to 2x-1$: `pol1_4 DUP + --;`
$x \to 10x-3$: `pol1_5 CONST 10 * -- -- --;`
x -> $11x + 7$: `pol1_6 CONST 11 * CONST 7 +;`
$x \to 2x^2$: `pol2_1 DUP OVER + *;`
$x \to -3x^2 + x$: `pol2_2 DUP CONST -3 * + + *;`
x -> $x^2 -2x + 1$: `pol2_3 -- DUP *;`
$(a,b,c) \to b^2-4ac$: `discr OVER ROT * -ROT * CONST 4 * -;`
Digits:
Lower digit: `digit0 CONST 10 %;`
Next digit: `digit1 CONST 10 / CONST 10 %;`
Hundreds: `digit2 CONST 10 DUP * / CONST 10 %;`
The highest digit: `digitH DUP C10 / DUP -ROT IF -5 SWAP DROP;`

Using genes. Using the constructed algorithms as "genes", we managed to find the following algorithms.

Sum of three squares, sorting/mid of three numbers, GCD in general case (arbitrary x,y), factorial (arbitrary x), left/right shift, minimal natural divider in general case (arbitrary x). Arbitrary decimal/hex/binary digits. The average of the three numbers. Binomial coefficients.

Here are, for example, some of these programs.

Right shift: $(n,x) \to 2^n x$ (only for $n>0$)
```
:    DUP + SWAP -- DUP -ROT IF -7 +;
```
Digit K: $(K, x) \to (x/10^K) \% 10$
```
:    digitK DUP IF 4 SWAP C10 / OVER --
     ROT IF -7 DROP C10 %;
```
GCD(x,y):
```
:    GCD_0 DUP -ROT % DUP IF -5 -;
:    GCD_1 OVER ROT IF 3 GOTO -5 GCD_0;
:    GCD ABS SWAP ABS GCD_1;
```

or, without genes:

```
:    GCD ABS SWAP ABS OVER ROT
     IF 3 GOTO -5 DUP -ROT % DUP
     IF -5 -;
:    FACTORIAL CONST 1 OVER -- -ROT *
     OVER IF -6 SWAP DROP;
:    BINOM DUP 2PICK — SWAP FACTORIAL
     SWAP FACTORIAL / SWAP FACTORIAL /;
```

Programs are given in full, with already built-in genes.

An exe-file that implements the algorithms described in the work can be obtained on our website [1]. A detailed description of the program is available in our text in arxiv.org [26] (this text contains a very detailed description of our programs which we removed from the present paper for the purpose of conciseness).

Conclusion

The proposed GP method allows to obtain programs that implement quite complex algorithms: factorial, binomial coefficients, search for a prime divider, etc. The achieved level of program complexity significantly exceeds that obtained by other methods. However, there are still many algorithms that cannot be implemented. For example, the finding of Fibonacci numbers is not yet obtained without "prompts", that is, without additional genes.

Based on the analysis of the obtained results, it is possible to build more advanced methods that can enable the implementation of even more complex algorithms.

References

1. **Gen_03.exe,** available at: http://math.ivanovo.ac.ru/dalgebra/Khashin/gene/gen_03r.htm (date:04.06.2018).

2. **genetic-programming.com**, available at: http://www.genetic-programming.com/ (date:04.06.2018).

3. **TensorFlow**: An open source machine learning framework for everyone, available at: https://www.tensorflow.org/ (date:04.06.2018).

4. **Thinking Forth Project,** available at: http://thinking-forth.sourceforge.net/ (date:30.06.2018).

5. **Wikipedia: Forth,** available at: URL:https://en.wikipedia.org/wiki/Forth_programming_language \\(date:30.06.2018).

6. **Wikipedia: LGP** Linear genetic programming, available at: https://en.wikipedia.org/wiki/Linear_genetic_programming \\ (date:04.06.2018).

7. **Koza, J. R.** Genetic programming as a means for programming computers by natural selection, *Stat Comput.*, 1994, vol. 4(2), pp. 87—112, doi:10.1007/BF00175355.

8. **Banzhaf W., Nordin P., Keller R. E., Francone F.** Genetic Programming — An Introduction; *On the Automatic Evolution of Computer Programs and its Applications*, M., Kaufmann Publ. Inc. CA, USA, 1998, 480 p.

9. **Panchenko T. V.** Genetic algorithms, Astrakhan Univ. publ., 2007, 87 p.

10. **Katayama S.** MagicHaskeller on the Web: Automated Programming as a Service, *In Haskell '13: Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell. ACM.*

11. **Francoso L., Sotto D. P., de Melo V. V.** Comparison of linear genetic programming variants for symbolic regression, *GECCO'14. Proc. 2014 Annual Conference on Genetic and Evolutionary Computation*, doi: 10.1145/2598394.2598472.

12. **Turner A. J., Miller J. F.** Recurrent Cartesian genetic programming applied to famous mathematical sequences, *Proceedings of the 7th York Doctoral ymposium on Computer Science&Electronics*, 2014, cs.york.ac.uk, pp. 37—47.

13. **Helmuth T., Spector L.** General Program Synthesis Benchmark Suite, *GECCO'15. Proc. 2015 Annual Conference on Genetic and Evolutionary Computation*, p. 1039—1046.

14. **Yalin K., Stolee Kathryn T., Le C., Brun Y.** Repairing Programs with Semantic Code Search., *ASE'15, Proceedings of the 2015 30th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE)*, pp. 295—306, doi: 10.1109/ASE.2015.60.

15. **Turner A. J., Miller J. F.** Neutral genetic drift: an investigation using Cartesian Genetic Programming, *Genet Program Evolvable Mach* (2015), doi:10.1007/s10710-015-9244-6.

16. **Sheppard C.** Genetic Algorithms with Python, 2016, 282 p.

17. **Chen Q., Xue B., Shang L., Zhang M.** Generalisation of Genetic Programming for Symbolic Regression with Structural Risk Minimisation, *GECCO '16 Proceedings of the Genetic and Evolutionary Computation Conference 2016* P. 709—716, doi:10.1145/2908812. 2908842.

18. **Sotto L. F. D. P., Melo V. V.** A Probabilistic Linear Genetic Programming with Stochastic Context-Free Grammar for solving Symbolic Regression problems, *arXiv:1704.00828* [cs.NE].

19. **Staats K., Pantridge E., Cavaglia M., Milovanov I., Aniyan A.** TensorFlow Enabled Genetic Programming, *arXiv:1708.03157* [cs.DC]

Simson J. Open-Source Linear Genetic Programming, *Ph.D. thesis, Waikato*, New Zealand.

21. **Pantridge Helmuth T., McPhee N., Spector L.** On the Diffculty of Benchmarking Inductive Program Synthesis Methods, *In Proceedings of GECCO '17 Companion*, Berlin, Germany, July 15—19, 2017, 8 p., doi:10.1145/3067695.3082533.

22. **Thi T. P., Nguyen X. H., Nguyen T. T.** A Study on Fitness Representation in Genetic Programming, *CTA 2016. Advances in Intelligent Systems and Computing*, vol 538, Springer, Cham, doi:10.1007/978-3-319-49073-113.

23. **Milano N., Nolfi S.** Scaling Up Cartesian Genetic Programming through Preferential Selection of Larger Solutions, *arXiv:1810.09485* [cs.NE].

24. **Wilson D. G., Miller J. F., Cussat-Blanc S., Luga H.** Positional Cartesian Genetic Programming, *arXiv:1810.09485* [cs.NE].

25. **Virgolin M., Alderliesten T., Witteveen C., Bosman P.** A Model-based Genetic Programming Approach for Symbolic Regression of Small Expressions, *arXiv:1904.02050* [cs.NE] (submitted for peer review to IEEE Transactions on Evolutionary Computation).

26. **Khashin S. I., Vaganov S. E.**, Genetic algorithms in Forth, *arXiv: 1807.06230* [math.NT].