Automated Design of a Neuroevolution Program Using Algebra-Algorithmic Tools

Anatoliy Doroshenko[1,2][0000-0002-8435-1451], Illia Achour[2][0000-0003-2348-8777], and Olena Yatsenko[1][0000-0002-4700-6704]

[1] Institute of Software Systems of National Academy of Sciences of Ukraine, Glushkov prosp. 40, 03187 Kyiv, Ukraine
[2] National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", 37, Prosp. Peremohy, Kyiv, 03056, Ukraine
doroshenkoanatoliy2@gmail.com, ilyaachour@gmail.com, oayat@ukr.net

Abstract. The adjustment of the previously developed algebra-algorithmic tools towards the automated design and generation of programs that use neuroevolutionary algorithms is proposed. Neuroevolution is a set of machine learning techniques that apply evolutionary algorithms to facilitate the solving of complex tasks, such as games, robotics, and simulation of natural processes. The developed program design toolkit provides automated construction of high-level algorithm specifications represented in Glushkov's system of algorithmic algebra and synthesis of corresponding programs based on implementation templates in a target programming language. The adjustment of the toolkit for designing neuroevolutionary algorithms consists in adding the descriptions and software implementations of the relevant elementary operators and predicates to a database of the toolkit. The use of the toolkit is illustrated by an example of designing and generating a program for the single-pole balancing problem, which applies the neuroevolutionary algorithm of the NEAT-Python library. The results of the experiment consisting in the execution of the program generated with the algebra-algorithmic toolkit are given.

Keywords: Automated Design, Algebra of Algorithms, Neuroevolution of Augmenting Topologies, Neural Network, Program Generation.

1 Introduction

Today, artificial neural networks are used in various areas of computer intelligence applications. Neural networks with a small number of neurons, due to their limited approximation capabilities, do not allow solving real practical tasks [1], and the selection of an excess number of neurons in the network leads to the problem of retraining and loss of approximation properties, an increase in the number of necessary hardware resources for its implementation, and an increase in time delays in processing information and training neural networks.

Traditionally, the structure of neural networks is determined manually by an expert developer of the computing systems that use these networks. Optimization of the structure of the neural network is carried out by removing some insignificant weighting coefficients. To implement this, algorithms of successive reduction or increase of the structure and methods of evolutionary optimization (neuroevolution) are used. The latter are applied to solve various tasks of neural network synthesis, namely: selection of features, adjustment of weights, selection of optimal network architecture, an adaptation of learning rules, and initialization of weight coefficient values.

The method of neuroevolution has become very common in recent years [2, 3]. Well-known AI laboratories and researchers are experimenting with it, some new successes are fueling enthusiasm, and new opportunities are emerging to influence deep learning, which has been and continues to be popular over the past decade. The neuroevolution method automates the creation of neural networks. Unlike manual methods, it allows for simultaneous changes in the weights of connections between nodes and the topology of nodes during the design process. One of the most famous implementations of neuroevolutionary algorithms is the NeuroEvolution of Augmenting Topologies (NEAT) [4, 5, 6]. Incremental topology neuroevolution is a form of reinforcement learning that uses evolutionary (genetic) algorithms to generate artificial neural networks, their parameters, topologies, and decision rules. The main advantage of neuroevolution lies in the possibility of its wider application compared to supervised learning, which requires marked correct pairs of input and output data for training. In contrast, neuroevolution requires only the ability to evaluate the performance of the generated network at any stage of training.

In this paper, facilities for automated design and generation of programs that use neuroevolutionary algorithms are proposed. The tools apply high-level specifications of programs based on Glushkov's systems of algorithmic algebras (SAA) [7, 8]. Based on the schemes represented in SAA, automated code generation is carried out in a target programming language. The approach is demonstrated on a program for single-pole balancing problem. The design and generation of the program using the open-source library for the implementation of neuroevolutionary algorithms NEAT-Python [9] were carried out.

2 Neuroevolution of Augmenting Topologies

Neuroevolution is a family of machine learning techniques that use evolutionary algorithms to facilitate solving complex tasks such as games, robotics, and modeling natural processes [6]. Neuroevolutionary algorithms imitate the process of natural selection. Very simple artificial neural networks can become very complex. The result of neuroevolution is an optimal network topology, which makes the model more energy efficient and convenient for analysis. The NEAT method is designed to reduce the dimensionality of the parameter search space in the form of a gradual development of the neural network structure in the process of evolution. The evolutionary process begins with a population of small, simplest genomes and gradually increases their complexity with each new generation.

Briefly, the essence of the NEAT approach is the following. In the beginning, we can have a very simple topology consisting only of input $(x_0, x_1, \ldots)$ and output (a)

nodes, as well as (optionally) a hidden layer (h) of nodes, which may be omitted. Next, a genetic algorithm works, which, through crossing and mutations, automatically produces other topologies, which are evaluated based on the fitness function, from which the best ones are selected to continue the generation of populations. At the same time, the quality of recognition improves due to increasing the number of hidden layers of the neural network. Quality can be monitored by observing a plot of the fitness function versus the number of populations generated. As experience shows, the best quality neural network is created as a result, which also has the advantages of compactness and speed.

One of the popular test tasks, which is effectively solved using NEAT, is pole balancing [3, 6, 10, 11]. The challenge is to steer a simulated cart that can only move in two directions using a pole attached to its top by a hinge (inverted pendulum). The longer the cart (controlled by a neural network) can hold the pole in the air, the higher its fitness. This task is very similar to trying to balance a pencil in the palm of a hand — it requires strong coordination and quick reaction.

Software implementations of the neuroevolution algorithm are represented by many libraries, in particular, NEAT-Python [9], SharpNEAT [12], PyTorch-NEAT [13], MultiNEAT [14], and NEAT Java [15]. In this work, an experiment was conducted to automate the development of the neuroevolution algorithm, which was implemented using the NEAT-Python library.

NEAT-Python is an implementation of the NEAT algorithm in the Python programming language. The NEAT-Python library provides an implementation of standard NEAT methods for modeling the genetic evolution of genomes of organisms in a population. It contains utilities for converting an organism's genotype to its phenotype (artificial neural network) and provides convenient methods for loading and saving the genome configuration together with NEAT parameters. In addition, it offers researchers useful routines that help to collect statistics about the progress of the evolutionary process and save/load intermediate control points. Control points allow periodic saving of the state of the evolutionary process and later resume the execution of the process from the saved control points.

The advantages of the NEAT-Python library include:

- stable implementation, comprehensive documentation;
- availability for easy installation using the PIP package manager;
- the presence of built-in statistics collection tools and support for saving execution checkpoints, as well as restoring execution from a given checkpoint;
- availability of several types of activation functions, support of continuous-time recurrent neural network phenotypes;
- easy extensibility to support different NEAT modifications.

The NEAT-Python library uses a set of hyperparameters that affect the performance and accuracy of the NEAT algorithms (the configuration file is stored in a format similar to Windows .ini files). Options include, but are not limited to:

- *fitness_criterion* — a function that computes the termination criterion from the set of fitness values of all genomes in the population;

- *fitness_threshold* — a threshold value that is compared with the fitness computed using the fitness_criterion function to check the need to complete the evolution;
- *pop_size* — the number of individual organisms in each generation;
- the initial configuration of the network by the number of hidden (*num_hidden*), input (*num_inputs*), and output nodes (*num_outputs*).

3 Facilities for Automated Design of Algorithms and Programs

To automate the development of programs, this work uses the SAA/1 language and the algebra-algorithmic Integrated toolkit for Designing and Synthesis of programs (IDS toolkit) [16, 17]. The language is intended for multi-level structural design and documentation of sequential and parallel algorithms and programs and is based on Glushkov's SAA [7, 8].

The main operator constructs of the SAA/1 language used in this work are the following:

- composition (sequential execution) of operators: "*operator*1"; "*operator*2" ;
- branching: IF '*condition*' THEN "*operator*1" ELSE "*operator*2" END IF ;
- for loop: FOR EACH ("*variable* " IN "*expression* ") "*operator*" END OF LOOP .

The process of automated development of algorithms and programs in the toolkit is shown in Fig. 1. The toolkit includes the following components:

- designer of high-level schemes of algorithms presented in SAA (SAA schemes);
- a database containing a description of SAA constructs and parameterized templates for implementing these constructs in target programming languages (C++, Java, Python);
- a generator of program texts based on algorithm schemes using templates from the database.

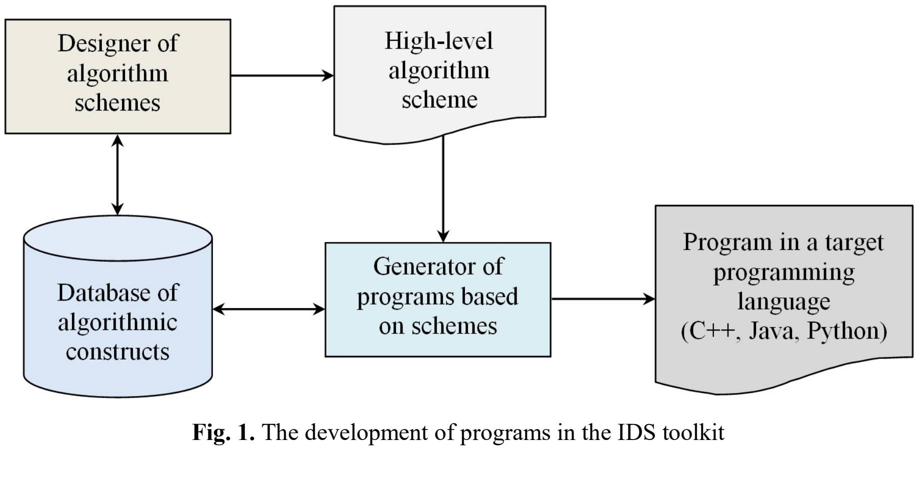
Designer of algorithm schemes
High-level algorithm scheme
Database of algorithmic constructs
Generator of programs based on schemes
Program in a target programming language (C++, Java, Python)

Fig. 1. The development of programs in the IDS toolkit

In the process of working with the designer, the user can edit the description of operator and predicate language constructs, as well as basic operators and conditions stored in the toolkit database.

The description of the database element (operation of SAA or a basic concept) includes its presentation in an analytical and natural-linguistic form, as well as its implementation in a programming language.

Setting up the toolkit for designing neuroevolutionary algorithms, which is the purpose of this work, consists in entering the descriptions of the corresponding basic operators and predicates into the database. Table 1 provides examples of the description of basic elements for neuroevolution algorithms. Identifiers of operators are enclosed in quotation marks, predicates are enclosed in apostrophes. The implementation in programming language uses methods of the NEAT-Python library.

Table 1. Examples of description of basic operators and predicates for neuroevolution tasks in the database of the IDS toolkit

No.	Natural-linguistic form	Implementation in Python language using the NEAT-Python library
1	"Activate the NET (net)(input)"	%1.activate(%2)
2	"Load configuration (config) from file (file)"	%1 = neat.Config(neat.DefaultGenome, neat.DefaultReproduction, neat.DefaultSpeciesSet, neat.DefaultStagnation, %2)
3	"Create the population (p), which is the top-level object for a NEAT run, for configuration (config)"	%1 = neat.Population(%2)
4	"Run neuroevolution for up to (n) generations and display the best genome (best_genome) among generations"	%2 = p.run(eval_genomes, %1=%1_generations) print('\nBest genome:\n{!s}'.format(%2))
5	"Set (genome) fitness to (value)"	%1.fitness = %2
6	"Adjust (fitness) based on (additional_num_runs) and (success_runs)"	%1 = 1.0 - (%2 - %3) / %2
7	"Create feedforward neural network (net) for (genome) and configuration (config)"	%1 = neat.nn.FeedForwardNetwork. create(%2, %3)
8	"Evaluate fitness of the genome that was used to generate net (net)"	fitness = cart.eval_fitness(%1)
9	'Genome (fitness) is greater or equal to fitness threshold (fitness_threshold) for configuration (config)'	%1 >= %3.%2
10	'Genome (fitness) is less than fitness threshold (fitness_threshold) for configuration (config)'	%1 < %3.%2

The basic operators for the implementation of neuroevolutionary algorithms include the following.

1. Activation of the neural network based on the array of input data.
2. Downloading the experiment configuration from a file.
3. Creation of the population p based on the configuration.
4. Starting the process of neuroevolution for n populations and displaying the best genome among populations.
5. Setting the genome fitness.
6. Adjusting the fitness value based on the number of additional simulation runs and successful runs.
7. Creation of a neural network for the specified genome and configuration.
8. Evaluation of the fitness of the genome that was used for network generation.

The predicates (numbers 9 and 10) are intended for checking the fitness value against the threshold value specified in the configuration file.

The natural-linguistic form of the description of basic elements contains the names of the formal parameters indicated in parentheses. The actual parameters specified in the SAA schemes replace the corresponding formal parameters in the text of the implementation of the basic concept in the programming language during program synthesis. Parameters in the implementation of the programming language are marked with the numbers %1, %2, etc.

An example of designing an application using some of the considered basic operators and predicates is given in the next section.

4 Application of the Integrated Toolkit for Designing a Program for Single-Pole Balancing Problem using the NEAT-Python Library

As an illustration, consider the use of the IDS toolkit for the well-known single-pole balancing problem [6]. This task is a classic example of a reinforcement learning experiment, which is also a commonly accepted benchmark for testing different implementations of control strategies.

In this experiment, the NEAT algorithm is used to implement a controller that regulates a physical device (a cart). The SAA facilities and the IDS toolkit are used to implement a cart device simulator and a neuroevolution-based control algorithm to stabilize the inverted pendulum over a given time interval. The NEAT-Python library [9] is used to implement the neuroevolutionary algorithm.

The statement of the single-pole balancing problem and the mathematical model (motion equations) are described in detail in [6]. A pole balancing controller takes scaled input values and produces an output signal that is a binary value and determines the action to be taken at a given time.

The initial configuration of the neural network of the controller is represented in Fig. 2. It includes five input nodes: for the horizontal position of the cart (x1) and its velocity (x2), for the vertical angle of the pole (x3) and its angular velocity (x4), and an additional input node for bias (x0) (which can be optional depending on the specific NEAT library used). The output node (a) is a binary node that outputs a control signal (0 or 1). The hidden node (h) is optional and can be omitted.

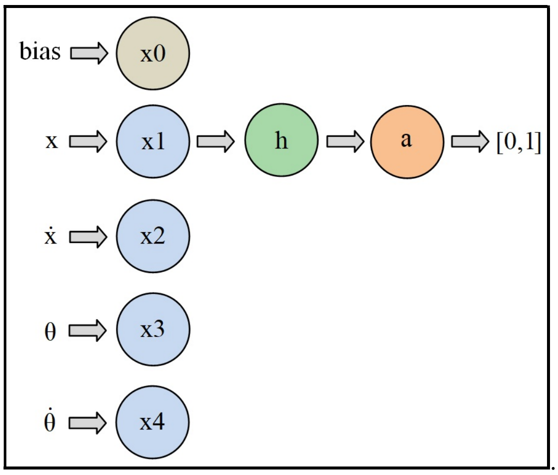

Fig. 2. The initial configuration of a neural network of a single-pole balancing controller

The simulation loop uses the controller's neural network to estimate the current state of the system and select the appropriate action (force to be added to the cart) for the next step. The neural network is created for each genome of the population at a certain generation of evolution, which allows evaluation of the efficiency of all genomes.

The simulation loop consists of the following steps.

1. Initialization of initial state variables (x, x_dot, theta, theta_dot) with random values within the limits.
2. Simulation loop through a certain number of steps specified by the max_bal_steps parameter.
3. The state variables are scaled to fit the range [0, 1] before loading them as inputs to the neural network controller.
4. The scaled input can be used to activate the neural network, and the output of the neural network is used to obtain a discrete action value.
5. After receiving the action values and the current values of the state variables, a single step of the cart-pole simulation can be run. After the simulation step, the returned state variables are checked against constraints to see if the system state is within acceptable limits.

The SAA scheme implementing the experiment is given below. The scheme begins with defining the libraries and constants used. Next is the compound main statement. The function of evaluating the suitability of all genomes in the population is contained in the compound operator "Evaluate the best net". It receives a list of all genomes in the population and NEAT configuration parameters. For each genome, it creates a neural network phenotype and uses it as a controller to run the simulation.

SCHEME SINGLE POLE EXPERIMENT

```
"GlobalData"
==== "Import library (os)";
     "Import library (neat)";
     "Import library (visualize)";
     "Import library (cart_pole as cart)";
     "Import library (utils)";
     "Set current working directory (local_dir)";
     "Set the directory to store outputs (out_dir) (local_dir) (out)";
     "Set the directory to store outputs (out_dir) (out_dir) (single_pole)";
      (additional_num_runs := 100);
      (additional_steps := 200)

"main"
==== "Determine path (config_path) to configuration file (local_dir)
      (single_pole_config.ini)";
     "Clean results of the previous run if any or init the output directory (out_dir)";
     "Run experiment (config_path)";

"Run experiment (config_file, n_generations = 100)"
==== "Load configuration (config) from file (config_file)";
     "Create the population (p), which is the top-level object for a NEAT run,
      for configuration (config)";
     "Add a stdout reporter to show progress in the terminal  for population (p)";
     "Run neuroevolution for up to (n) generations and display the best
      genome (best_genome) among generations";
     "Create feedforward neural network (net) for (best_genome) and configuration
      (config)";
     "Output the message (\n\nEvaluating the best genome in random runs)";
     (success_runs := "Evaluate the best net (net, config, additional_num_runs)");
     "Output successful (success_runs) and expected runs (additional_num_runs)
      and check if the best genome is a winner";
     "Visualize the experiment results for configuration (config), best genome
      (best_genome) from directory (out_dir)";

"Evaluate genomes (genomes, config)"
==== FOR EACH (genome_id, genome IN genomes)
       "Set (genome) fitness to (0.0)";
       "Create feedforward neural network (net) for (genome) and
        configuration (config)";
       "Evaluate fitness of the genome that was used to generate net (net)";
```

```
    IF ‘Genome (fitness) is greater or equal to the fitness threshold
         (fitness_threshold) for configuration (config)’
    THEN
      (success_runs := “Evaluate the best net (net, config,
                              additional_num_runs)”);
      “Adjust (fitness) based on (additional_num_runs) and (success_runs)”
    END IF;
    “Set (genome) fitness to (fitness)”
  END OF LOOP;

“Evaluate the best net (net, config, num_runs)”
==== FOR EACH (run IN range(num_runs))
    “Evaluate fitness of the genome that was used to generate net (net,
      max_bal_steps = additional_steps)”
    IF ‘Genome (fitness) is less than fitness threshold (fitness_threshold) for
         configuration (config)’
    THEN “Return value (run)”
    END IF;
  END OF LOOP;
  “Return value (num_runs)”

“Evaluate fitness of the genome that was used to generate net (net,
max_bal_steps = 500000)”
==== (steps := “Run cart-pole simulation (net, max_bal_steps)”);
  IF (steps = max_bal_steps)
  THEN “Return value (1.0)”
  ELSE IF (steps = 0)
       THEN “Return value (0.0)”
       ELSE
         (log_steps := log(steps));
         (log_max_steps := log(max_bal_steps));
         “(error) := ((log_max_steps – log_steps) / log_max_steps)”;
         “Return value (1.0 – error)”
       END IF
   END IF

END OF SCHEME SINGLE POLE EXPERIMENT
```

The function of performing the experiment is in the compound statement “Run experiment”. It starts by loading hyperparameters from a configuration file and generates the initial population using the loaded configuration. The function then configures reporters to collect statistics related to the execution of the evolutionary process. Output reporters are also added to write the execution results to the console in real time. The evolution process is performed for the specified number of generations, and the results are stored in the source directory. After the best genome has been found during the evolutionary process, a check is made to see if it meets the fitness threshold criteria set in the configuration file. The program returns the genome with the formal best match.

5 Experimental Results

Now, based on the designed SAA scheme, the generation of the program code in the Python language can be automated using the IDS toolkit. Below, there are the results of the experiment on the execution of the generated program. A population of 150 individual organisms was defined in the NEAT-Python configuration file and a fitness threshold of 1.0 was set as the termination criterion. The values of the initial parameters of the neural network are the following: *num_hidden* = 0, *num_inputs* = 4, and *num_outputs* = 1.

When the program is executed, data is displayed for each generation of evolution. The best genome, which is the winner of evolution, encodes a neural network phenotype consisting of only one nonlinear node (output) and three connections from input nodes (size: (1, 3)). The graph of the neural network of the winning controller for the single-pole balancing problem is shown in Fig. 3.

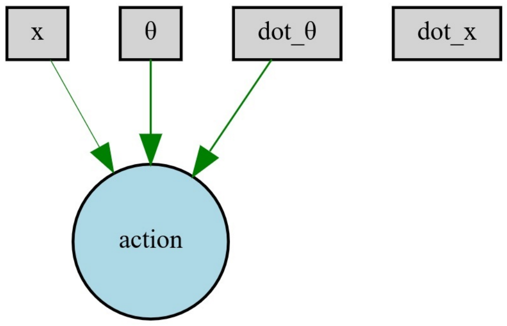

Fig. 3. Graph of the optimal single-pole balancing controller found by the NEAT algorithm

The graph of changes in fitness values over generations of evolution is shown in Fig. 4.

The average fitness of the population in all generations was low, but from the beginning, there was a beneficial mutation that gave rise to a certain line of organisms. From generation to generation, gifted individuals from this line were able not only to preserve their useful traits but also to improve them, which ultimately led to the emergence of an evolutionary winner.

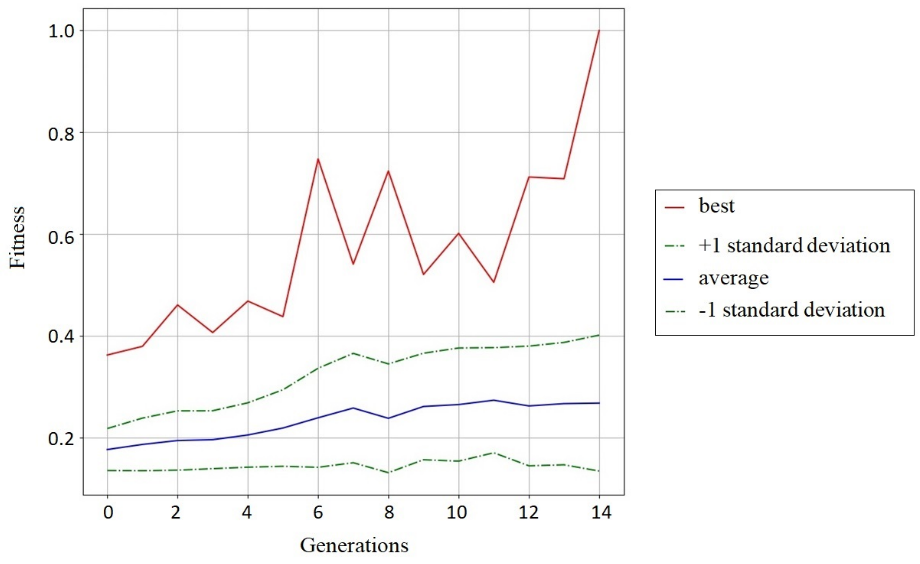

Fig. 4. Average and best values of the fitness function in the single-pole balancing experiment

6 Conclusion

The adjustment of the previously developed algebra-algorithmic toolkit towards the automated design and synthesis of programs using neuroevolutionary algorithms was proposed. The method of neuroevolution of augmenting topologies is intended to reduce the dimensionality of the space for searching for neural network parameters in the form of gradual development of the neural network structure in the process of evolution. The basis of the toolkit is the construction of high-level specifications of algorithms represented in systems of algorithmic Glushkov's algebras, and the generation of corresponding programs based on implementation templates in a target programming language. The approach is illustrated by the example of designing a program for the single-pole balancing problem, which uses the neuroevolutionary algorithm of the NEAT-Python library.

References

1. Subbotin, S.O., Oliinyk, A.O., Oliinyk, O.O.: Non-iterative, evolutionary and multi-agent methods for synthesis of fuzzy and neural network models. ZNTU, Zaporizhzhia (2009). (in Ukrainian)
2. Stanley, K.O., Clune, J., Lehman, J., Miikkulainen, R.: Designing neural networks through neuroevolution. Nature Machine Intelligence 1, 24–35 (2019). https://doi.org/10.1038/s42256-018-0006-z
3. Stanley, K.O.: Neuroevolution: a different kind of deep learning, https://www.oreilly.com/radar/neuroevolution-a-different-kind-of-deep-learning, last accessed 2023/05/05

4. Doroshenko, A.Y., Achour, I.Z., Yatsenko, O.A.: Parameter-driven generation of evaluation program for a neuroevolution algorithm on a binary multiplexer example. Radio Electronics, Computer Science, Control (1), 80–88 (2023). https://doi.org/10.15588/1607-3274-2023-1-8
5. Stanley, K.O., Miikkulainen, R.: Evolving neural networks through augmenting topologies. Evolutionary Computation 10(2), 99–127 (2002). https://doi.org/10.1162/10636560 2320169811
6. Omelianenko, I.: Hands-on neuroevolution with Python. Packt, Birmingham (2019).
7. Doroshenko, A., Yatsenko, O.: Formal and adaptive methods for automation of parallel programs construction: emerging research and opportunities. IGI Global, Hershey (2021). http://doi.org/10.4018/978-1-5225-9384-3
8. Andon, P.I., Doroshenko, A.Yu., Zhereb, K.A., Yatsenko, O.A.: Algebra-algorithmic models and methods of parallel programming. Akademperiodyka, Kyiv (2018). https://doi.org/10.15407/akademperiodyka.367.192
9. NEAT-Python, https://github.com/CodeReclaimers/neat-python, last accessed 2023/05/05
10. Chang, O., Kwiatkowski, R., Chen, S., Lipson, H.: Agent embeddings: a latent representation for pole-balancing networks. In: AAMAS'19, pp. 656–664. International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC (2019). https://doi.org/10.48550/arXiv.1811.04516
11. Lawrence, W.M.: Solving XOR and Pole-balancing problems using a multi-population NEAT. A thesis presented for the degree of Master of Philosophy. De Montfort University, Leicester (2020). https://dora.dmu.ac.uk/bitstream/handle/2086/20731/William-Lawrence.pdf?sequence=1
12. SharpNEAT. Evolution of Neural Networks, https://sharpneat.sourceforge.io, last accessed 2023/05/05
13. PyTorch-NEAT, https://github.com/uber-research/PyTorch-NEAT, last accessed 2023/05/05
14. MultiNEAT: Portable NeuroEvolution Library, https://github.com/peter-ch/MultiNEAT, last accessed 2023/05/05
15. NEAT Java (JNEAT), https://nn.cs.utexas.edu/soft-view.php?SoftID=5, last accessed 2023/05/05
16. Doroshenko, A., Shymkovych, V., Yatsenko, O., Mamedov, T.: Automated software design for FPGAs on an example of developing a genetic algorithm. In: Burov, O., Ignatenko, O., Kharchenko, V., Kobets, V. et al. (eds.) ICTERI 2021, CCIS, vol. 1635, pp. 74–85. Springer, Cham (2021).
17. Doroshenko, A., Zhereb, K., Yatsenko, O.: Using algebra-algorithmic and term rewriting tools for developing efficient parallel programs. In: Ermolayev, V., Mayr, H.C., Nikitchenko, M., Spivakovsky, A. (eds.) ICTERI 2013, CCIS, vol. 1000, pp. 38–46. Springer, Cham (2013).