ПРОГРЕСИВНІ ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ

PROGRESSIVE INFORMATION TECHNOLOGIES

UDC 004.4'24, 004.896

PARAMETER-DRIVEN GENERATION OF EVALUATION PROGRAM FOR A NEUROEVOLUTION ALGORITHM ON A BINARY MULTIPLEXER EXAMPLE

Doroshenko A. Yu. – Dr. Sc., Professor, Head of the Computing Theory Department, Institute of Software Systems of the National Academy of Sciences of Ukraine, Kyiv, Ukraine.

Achour I. Z. – Post-graduate student of the Department of Information Systems and Technologies, National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", Kyiv, Ukraine.

Yatsenko O. A. – PhD, Senior Researcher of the Computing Theory Department, Institute of Software Systems of the National Academy of Sciences of Ukraine, Kyiv, Ukraine.

ABSTRACT

Context. The problem of automated development of evaluation programs for the neuroevolution of augmenting topologies. Neuroevolution algorithms apply mechanisms of mutation, recombination, and selection to find neural networks with behavior that satisfies the conditions of a certain formally defined problem. An example of such a problem is finding a neural network that implements a certain digital logic.

Objective. The goal of the work is the automated design and generation of an evaluation program for a sample neuroevolution problem (binary multiplexer).

Method. The methods and tools of Glushkov's algebra of algorithms and hyperscheme algebra are applied for the parameter-driven generation of a neuroevolution evaluation program for a binary multiplexer. Glushkov's algebra is the basis of the algorithmic language intended for multilevel structural design and documentation of sequential and parallel algorithms and programs in a form close to a natural language. Hyperschemes are high-level parameterized specifications intended for solving a certain class of problems. Setting parameter values and subsequent interpretation of hyperschemes allows obtaining algorithms adapted to specific conditions of their use.

Results. The facilities of hyperschemes were implemented in the developed integrated toolkit for the automated design and synthesis of programs. Based on algorithm schemes, the system generates programs in a target programming language. The advantage of the system is the possibility of describing algorithm schemes in a natural-linguistic form. An experiment was conducted consisting in the execution of the generated program for the problem of evaluating a binary multiplexer on a distributed cloud platform. The multiplexer example is included in SharpNEAT, an open-source framework that implements the genetic neuroevolution algorithm NEAT for the .NET platform. The parallel distributed implementation of the SharpNEAT was proposed in the previous work of the authors.

Conclusions. The conducted experiments demonstrated the possibility of the developed distributed system to perform evaluations on 64 cloud clients-executors and obtain an increase in 60–100% of the maximum capabilities of a single-processor local implementation.

KEYWORDS: algebra of algorithms, automated program design, cloud computing, hyperscheme, neuroevolution, neural network, parallel programming.

ABBREVIATIONS

IDS is an Integrated toolkit for software Design and Synthesis;

NEAT is NeuroEvolution of Augmenting Topologies;

SAA is a system of algorithmic algebra;

SharpNEAT is an open-source framework written in C# that implements the genetic neuroevolution algorithm NEAT.

NOMENCLATURE

A is a nonterminal operator from set R_T;

A_j is an operator from set $\overline{Op}$;

AHS is algebra of hyperschemes;

e is an empty word;

E_3 is a three-valued logic;

E_4 is a four-valued logic;

$F(A, p)$ is a function that specifies the generation method for operations of AHS signature;

GA is Glushkov's algebra (system of algorithmic algebra);

IS is a set of states (an information set) of the operational automaton of the abstract automaton model of a computer;

$\overline{L}$ is a set of states of tape $\widetilde{L}$;

$\widetilde{L}$ is a tape of operational automaton $\overline{\Phi}$;

DOI 10.15588/1607-3274-2023-1-8

m is a number of address inputs of a multiplexer;

$\overline{M}$ is a set of states of operational automaton $\overline{\Phi}$;

$\widetilde{M}$ is a stack of control automaton $\overline{\Psi}$;

n is a number of data inputs of a multiplexer;

Op is a set of operators of GA;

$\overline{Op}$ is a set of operators of AHS;

p is an element of set $\overline{P}$;

$\overline{P}$ is a set of states (an information set) of the operational automaton of the abstract automaton model of the parameter-driven generator of texts;

P_0 is an array length;

P_1 is a number of address inputs of a multiplexer (hyperscheme parameter);

P_2 is a number of information inputs of a multiplexer (hyperscheme parameter);

P_3 is the total number of inputs of a multiplexer (hyperscheme parameter);

P_q is a hyperscheme parameter with number q;

Pr is a set of predicates of GA;

$\overline{Pr}$ is a set of predicates of AHS;

R_N is a set of nonterminal operators of AHS;

R_T is a set of terminal operators of AHS;

s_j is address input of a multiplexer;

u_k is a predicate from set $\overline{Pr}$;

x_i is a data input of a multiplexer;

y is an output of a multiplexer;

η is "not computed" value;

μ is "undefined" value;

$\overline{\Phi}$ is an operational automaton;

$\overline{\Psi}$ is a control automaton;

Ω_{AHS} is a signature of operations of AHS;

Ω_{GA} is a signature of operations of GA;

Ω_1 is a set of logic operations included in Ω_{GA};

Ω_2 is a set of operator operations included in Ω_{GA}.

INTRODUCTION

One of the promising directions in the development and research of parallel and distributed computing systems is the construction of software abstractions in the form of algebraic-algorithmic languages and models, which aims to develop architecture- and language-independent programming tools for multiprocessor computing systems and networks. In [1], authors proposed a theory, methodology, and software tools for the automated design of parallel programs based on high-level algebraic formalization and automation of program transformations based on rewriting rules. In particular, an instrumental system of programming automation called the integrated toolkit for software design and synthesis (IDS) was developed. One of the important problems within the algebra-algorithmic approach is increasing the adaptability of programs to the specific conditions of their use. In particular, it can be solved by using the method of parameter-driven generation of algorithm schemes based on higher-level specifications named hyperschemes.

In this paper, the developed algebra-algorithmic facilities are applied to the field of neuroevolution algorithms. Neuroevolution is a promising approach for solving complex problems of machine learning, the development of artificial neural networks, adaptive control, multi-agent systems, and evolutionary robotics [2]. The main advantage of neuroevolution is that it can be used more widely than supervised learning algorithms, which require a program of correct input-output pairs. Neuroevolution only requires evaluating the performance of the network when performing a task. It uses evolutionary algorithms to train a neural network and belongs to the category of reinforcement learning methods. All evolutionary algorithms develop a set ("population") of solutions ("individuals"). Individuals are represented by their genotype, which is expressed in the form of a phenotype, with which quality, "adaptability" is associated. There are a large number of neuroevolutionary algorithms, divided into two groups. The first includes algorithms that perform the evolution of weights for a given network topology, the second includes algorithms that, in addition to the evolution of weights, also perform the evolution of the network topology. Evolutionary algorithms manipulate a set of genotypes, which are a representation of a neural network. In a direct coding scheme, the genotype is equivalent to the phenotype, and neurons and connections are directly specified in the genotype. Conversely, in the scheme with indirect coding, the rules and structures for creating a neural network are specified in the genotype.

The object of study is the automated development of evolutionary algorithms.

One of the implementations of evolutionary algorithms is SharpNEAT [3], an open-source framework developed in the C# language. It implements the genetic neuroevolution algorithm NEAT (NeuroEvolution of Augmenting Topologies) for the .NET platform. The algorithm uses the evolutionary mechanisms of mutation, recombination, and selection to find neural networks with behavior that satisfies the conditions of a certain formally defined problem. Examples of such problems are controlling the movements of a robot's limbs, flying a rocket, or finding a neural network that implements a certain digital logic (for example, a multiplexer).

Despite the strengths of the NEAT method, such as the possibility of its application in tasks where it is difficult to choose the cost function and neural network topology, one of the problems is the slow convergence to optimal results, especially in the case of complex environments. The distributed implementation of the NEAT evaluation method was proposed in the previous work of the authors [5]. It allows to radically speed up finding optimal configurations of neural networks in the presence of sufficient computing resources.

DOI 10.15588/1607-3274-2023-1-8

The subject of study is the automated design and generation of evaluation programs for neuroevolution algorithms.

The purpose of the work is to apply algorithm algebra and hyperschemes [1, 6] for the parameter-driven generation of an evaluation program for a sample neuroevolution problem.

Hyperschemes are parameterized specifications intended for solving a certain class of problems. Setting specific values of parameters and subsequent interpretation of hyperschemes allows obtaining algorithms adapted to specific conditions of their use. The generator of algorithms based on hyperschemes is one of the components of the above-mentioned IDS toolkit [1]. Algorithm schemes being designed in the toolkit are presented in Glushkov's system of algorithmic algebra (SAA).

The approach to the parameter-driven generation of programs is illustrated on generating the source code of the evaluation procedure for the binary multiplexer problem example included in SharpNEAT [4]. The results of the execution of multi-threaded and distributed versions of the generated procedure on a multicore processor and a cloud platform are given.

1 PROBLEM STATEMENT

The problem consists in designing a high-level parameterized specification in the algebra of hyperschemes [1, 6] that is applied to generate classes of evaluation schemes for a binary multiplexer (Binary MultiplexerEvaluator) example [4] depending on the multiplexer parameters, followed by the automated synthesis of code in C# language for the SharpNEAT framework.

A multiplexer is a device that has several data inputs x_i $(i = 0,...,n-1)$, address inputs s_j $(j = 0,...,m-1)$, and one output y. The device transmits a signal from one of the data inputs to the output; at the same time, the selection of the desired input is carried out by applying the appropriate combination of control signals to the address inputs. The number of data inputs n and the number of address inputs m are related by the ratio: $n = 2^m$. The conditional scheme of the multiplexer with 11 inputs is shown in Fig. 1.

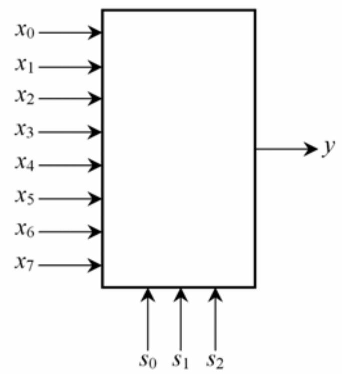

Figure 1 – The conditional scheme of a multiplexer with 11 inputs

The parameters of the hyperscheme are the number P_1 of address inputs of the multiplexer, the number P_2 of its information inputs and the total number of inputs $P_3 = P_1 + P_2$. All inputs accept binary values (0 or 1). A binary address is applied to the address inputs, representing the selection of one of the input values for data. The evaluation consists of exhaustively testing the neural network on each of the 2^{P3} possible input combinations [4]. The output value of the neural network must match the value of one of the data inputs, which is represented by a binary address from the address inputs. An output value less than 0.5 is considered a binary zero, and an output value greater than or equal to 0.5 is a binary one. The value of the assessment (suitability) is calculated additively as a result of the comprehensive check.

Depending on the values of the hyperscheme parameters, a specific scheme of an algorithm in SAA [1] is to be generated, representing a multiplexer evaluation scheme with a specific number of inputs. The examples of parameter values are shown in Table 1. The SAA schemes are the basis for the generation of C# programming code.

Table 1 – The values of the hyperscheme parameters (P_1–P_3) for generating multiplexer evaluation schemes

Number of multiplexer inputs	Corresponding values of hyperscheme parameters P_1	P_2	P_3
3	1	2	3
6	2	4	6
11	3	8	11

2 REVIEW OF THE LITERATURE

This paper is related to works on the automated generation of programs from specifications [7–10] and neuroevolution of augmenting topologies [2, 11, 12].

In particular, paper [7] presents a tool for the automatic generation of C++ programs from Isabelle (high-order logic theorem prover) specifications. In [8], a combination of code and test generation based on the specification language of the Temporal Logic of Actions (TLA) is proposed. Work [9] presents a tool for generating C++ code from abstract state machine models. Paper [10] proposes an automated technique that generates executable tests from structured natural language specifications.

The peculiar feature of our approach to program generation consists in using algebra of algorithms and hyperschemes [6]. Algorithms and programs are constructed using high-level algebra-algorithmic schemes represented in a natural linguistic form. The developed tools provide automated generation of sequential and parallel code in C++ and Java languages from the schemes. In this paper, we apply these algebra-algorithmic facilities for the automated design of an evaluation procedure for a neuroevolution algorithm.

Neuroevolution of augmenting topologies is a genetic algorithm for finding artificial neural networks through evolution (neuroevolutionary method) [2]. HyperNEAT (Hypercube-based NeuroEvolution of Augmenting To-

DOI 10.15588/1607-3274-2023-1-8

pologies) is an extension of NEAT that uses a form of indirect encoding called Compositional Pattern-Producing Networks (CPPNs) [11]. The implementations of NEAT and HyperNEAT are part of a package called SharpNEAT developed in C# by Colin Green [12]. The peculiarity of NEAT and SharpNEAT is that they search both the structure of the neural network (nodes and connections) and the weight parameters of connections between nodes. The parallel distributed version of SharpNEAT was proposed by the authors in [5].

In this work, the distributed version is applied for evaluating the performance of the code generated for binary multiplexer problem example on a cloud platform.

3 MATERIALS AND METHODS

In this section, we consider the system of algorithmic algebra and hyperschemes, which are the basis of the algebra-algorithmic approach to algorithm design and synthesis. The software tools for the automated generation of algorithm schemes and programs are also described.

Glushkov's SAA is focused on the analytical form of algorithm representation and formalized transformation of these specifications, in particular, with the aim of optimizing the algorithms according to specified criteria [1]. SAA is the two-sorted algebra $GA = \langle\{Pr, Op\}; \Omega_{GA}\rangle$, where sorts are a set Pr of predicates and a set Op of operators defined on information set IS. The operators are mappings (possibly partial) of IS to itself. The predicates take values of the three-valued logic $E_3 = \{0, 1, \mu\}$. The signature $\Omega_{GA} = \Omega_1 \cup \Omega_2$ consists of system Ω_1 of logic operations (conjunction, disjunction, negation, and prognosis) that take values in set Pr, and system Ω_2 of operator operations (composition, branching, loop, and other) that take values in set Op and are considered further in more detail.

SAA is the basis of the algorithmic language SAA/1, designed for multilevel structural design and documentation of sequential and parallel algorithms and programs. The advantage of its use is the possibility of describing algorithms in a natural-linguistic form. The operators represented in the SAA/1 language are called SAA schemes. Identifiers of predicates in this language are enclosed in single quotes, and operators – in double ones. Predicates and operators in SAA/1 can be basic or compound. Basic elements are elementary atomic abstractions in algorithm schemes. Compound conditions and operators are built from basic ones using the operations from the SAA signature.

Some operator operations of SAA used in this paper are the following (represented in a natural-linguistic form):

– composition (sequential execution) of operators: "operator1"; "operator2";

– branching: IF 'condition' THEN "operator1" ELSE "operator2" END IF;

– for loop: FOR (counter FROM start TO fin) "operator" END OF LOOP;

– parallel processing of a list: PARALLEL FOR EACH (elem IN list)("operator(elem)").

The algebraic facilities for generation of algorithm schemes are based on SAA and the abstract automaton model of the parameter-driven text generator [1, 6]. The generator works according to a feedback principle (see Fig. 2). The automaton $\overline{\Psi}$ with stack $\widetilde{M}$ is used as a control automaton, and the automaton $\overline{\Phi}$ with tape $\widetilde{L}$ is used as an operational one. Tape $\widetilde{L}$ is intended for writing the text of an SAA scheme being generated.

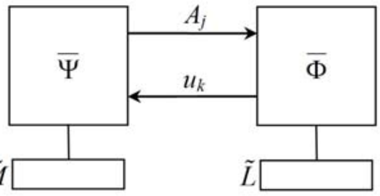

Figure 2 – The abstract automaton model of the parameter-driven text generator

Set $\overline{M}$ of states of automaton $\overline{\Phi}$ is associated with parameters that control the generation of schemes. The elements of the information set $\overline{P} = \overline{M} \times \overline{L}$ are called the states of the operational structure. At each step of the automaton's work, a set of values of logical conditions $\overline{Pr} = \{u_k\}$ defined on set $\overline{P}$ is sent from the operational to the control automaton. Depending on these values and contents of stack $\widetilde{M}$, the control automaton initiates the execution of some operator. The set of operators $\overline{Op} = \{A_j\}$ is divided into two disjoint sets – terminal operators R_T and nonterminal operators R_N. Execution of the terminal operator from set R_T consists in changing the current state of the operational structure, which, in particular, can be writing some text on tape $\widetilde{L}$. The execution of operator $A \in R_N$ at current state $p \in \overline{P}$ consists in writing some term $F(A, p)$ to stack $\widetilde{M}$ and its further interpretation by the control automaton. The term $F(A, p)$ is an analog of the concepts of macro definition, procedure, routine, etc. Stack $\widetilde{M}$ is used at processing nested and recursive terms. The generated text is the content of tape $\widetilde{L}$ in the final state of the operational structure

The considered abstract automaton model is matched with the algebra of hyperschemes intended for the formalization of algorithms for the parameter-driven generation of SAA schemes [6]. It is the two-sorted algebra $AHS = \langle\{\overline{Pr}, \overline{Op}\}; \Omega_{AHS}\rangle$, where predicates from set $\overline{Pr}$ are defined on information set $\overline{P}$ and take values of the four-valued logic $E_4 = \{0, 1, \mu, \eta\}$; operators from set $\overline{Op}$ are defined on and take values in set $\overline{P}$.

DOI 10.15588/1607-3274-2023-1-8

The set of predicates is associated with parameters that control the process of SAA scheme generation. The operations of the signature Ω_{AHS} are similar to the SAA operations. The difference from SAA is that the predicates from set $\overline{Pr}$ map information set $\overline{P}$ to set E_4 with additional value η, which is used to indicate that the value of a predicate cannot be computed due to a lack of information about the values of hyperscheme parameters.

The application of operator $A \in \overline{Op}$ at state $p \in \overline{P}$ leads to the transition of operational structure $\overline{\Phi}$ to a new state $A(p) \in \overline{P}$ and writing some (possibly empty) fragment $F(A, p)$ of a scheme being generated to tape $\tilde{L}$. The function $F(A, p)$ specifies the generation method for all operations of the algebra of hyperschemes and is defined in detail in [6].

In particular, function $F(A, p)$ for operation "operator1"; "operator2" generates the composition operation without changes.

For the operation of branching, the generation function is

$$F(A, p) = \begin{cases} \text{"operator1", if 'condition'} = 1; \\ \text{"operator 2", if 'condition'} = 0; \\ \text{IF 'condition' THEN "operator1" ELSE} \\ \text{"operator2" END, if 'condition'} = \eta; \\ e, \text{if 'condition'} = \mu, \end{cases}$$

where e is an empty word.

The result of the interpretation of this operation is the text of the first operator at the true value of the condition, and the text of the second operator at the false value. The whole text of the branch operation is generated at a not computed value of the condition. An empty text is a result in the case if there was an error during the interpretation process.

Representations of operators in AHS are called hyperschemes. Each hyperscheme A applied at state $p \in \overline{P}$ generates an SAA scheme $F(A, p)$. Hyperscheme A defines the class of SAA schemes $\{F(A, p) \mid p \in \overline{P}\}$.

The processing of basic conditions and operators of a hyperscheme consists in computing expressions with hyperscheme parameters and substituting them into the text of these basic elements.

The considered approach to the generation of algorithm schemes is implemented in the integrated toolkit for software design and synthesis [1]. Hyperschemes are designed in an automated way by detailing the language constructs of the hyperscheme algebra. The constructs are selected from a list and added to the algorithm design tree. At each step of the design process, the system offers a list of algebra operations depending on the type of tree node selected. A hyperscheme is used for further generation of an SAA scheme of an algorithm (see Fig. 3) and synthesis of a program in a target programming language (C, C++, Java, and other).

To facilitate processing, the parameters are written in the text of the basic and other elements of a hyperscheme in the form P_q $(q = 0, 1, 2, ...)$. Expressions with hyperscheme parameters are enclosed in square or curly brackets.

Example. Consider the use of the hyperscheme facilities for designing a fragment of the hybrid sorting algorithm.

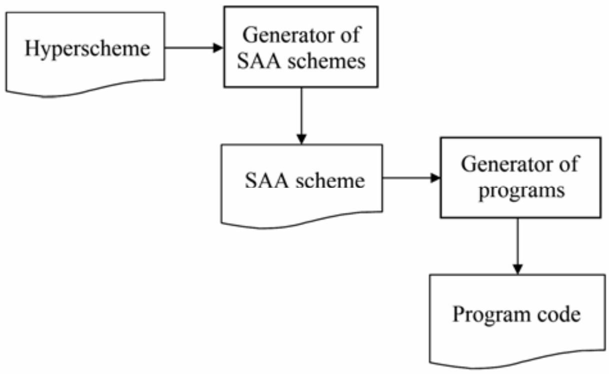

Figure 3 – The sequence of generation of algorithms and programs in the IDS toolkit

In the SAA scheme below, one of the sorting algorithms (insertion, sequential, or parallel merge sort) is selected depending on the length P_0 of the array.

```
"Hybrid sort (array)" ===
= IF '[P0 <= 200]'
  THEN
    "Insertion sort (array)"
  ELSE
    IF '[P0 <= 1000]'
    THEN
      "Sequential merge sort (array)"
    ELSE
      "Parallel merge sort (array)"
    END IF
  END IF
```

Let it be known in advance that the algorithm represented by the scheme will be applied in conditions when $P_0 \geq 500$, then the given SAA scheme becomes redundant. Considering it as a hyperscheme, we can assume that at the stage of generation of an SAA scheme, the condition '[P_0 <= 200]' takes the value "false", while '[P_0 <= 1000]' takes the value "not computed". As a result of the generation of text according to the hyperscheme, we will get the shortened SAA scheme:

```
"Hybrid sort (array)" ===
= IF '[P0 <= 1000]' THEN
    "Sequential merge sort (array)"
  ELSE
    "Parallel merge sort (array)"
  END IF
```


DOI 10.15588/1607-3274-2023-1-8

4 EXPERIMENTS

In this paper, we apply the facilities of hyperschemes for generating classes of SAA schemes intended for the evaluation of a binary multiplexer (BinaryMultiplexer Evaluator) [4].

The hyperscheme constructed using the IDS toolkit is given below. Its parameters P_1, P_2, P_3 were described in Section 1. In the scheme, curly brackets $\{P_3\}$ indicate the parameter that needs to be replaced with the corresponding number written in words, that is, if the value of $P_3 = 11$, the text "Eleven" will be inserted. Square brackets (for example, $[P_1]$ or $[P_3 - 1]$) indicate parameters or arithmetic expressions to be replaced by the corresponding number. So, for the loop FOR (i FROM 0 TO [Pow(2, P_3) – 1]) at the value of the parameter $P_3 = 11$, the text FOR (i FROM 0 TO 2047) will be generated.

```
SCHEME
BINARY{P3}MULTIPLEXEREVALUATOR ===
   "Binary {P3}-multiplexer evaluator scheme"
  END OF COMMENTS

"Binary{P3}MultiplexerEvaluator" ===
= NAME SPACE
  SharpNeat.Domains.Binary{P3}Multiplexer
  (
    CLASS Binary{P3}MultiplexerEvaluator OF
    TYPE public INHERITS
    IPhenomeEvaluator<IBlackBox>

      "Declare a constant (StopFitness) of type
       (double) = (10E + [P1])";
      "Declare a variable (_evalCount) of type
       (ulong)";
      "Declare a variable (_stopConditionSatisfied)
       of type (bool)";

     REGION IPhenomeEvaluator<IBlackBox>
               Members

       PROPERTY public ulong EvaluationCount
       GET
       (
         "Return value (_evalCount)"
       )
       END OF PROPERTY

       PROPERTY public bool
                  StopConditionSatisfied
       GET
       (
         "Return value (_stopConditionSatisfied)"
        )
       END OF PROPERTY

       METHOD public FitnessInfo
       Evaluate(IBlackBox box)
         "Declare a variable (fitness) of type
          (double) = (0.0)";
         "Declare a variable (success) of type
          (bool) = (true)";
         "Declare a variable (output) of type
          (double)";
         "Declare a variable (inputArr) of type
           (ISignalArray) = (box.InputSignalArray)";
         "Declare a variable (outputArr) of type
           (ISignalArray) = (box.OutputSignalArray)";
         "Increase (_evalCount) by (1)";
         FOR (i FROM 0 TO [Pow(2, P3) – 1])
         LOOP
           "Declare a variable (tmp) of type
             (int) = (i)";
           FOR (j FROM 0 TO [P3 – 1])
           LOOP
             (inputArr[j] := tmp&0x1);
             (tmp := tmp >> 1)
           END OF LOOP;
           "Activate the black box (box)";
           "Read output signal (output)(outputArr)";
           IF (((1 << ([P1] + (i&0x[P2 – 1])))&i) != 0)
           THEN
             (fitness := fitness + 1.0 – ((1.0 – output) *
                        (1.0 – output)));
             IF (output < 0.5)
             THEN (success := false)
             END IF
           ELSE
             (fitness := fitness + 1.0 – (output *
                        output));
               IF (output >= 0.5)
               THEN (success := false)
               END IF
           END IF;
           "Reset black box state ready for next test
             case (box)"
         END OF LOOP;
         IF success
         THEN (fitness := fitness + 10E + [P1])
         END IF;
         IF (fitness >= StopFitness)
         THEN (_stopConditionSatisfied := true)
         END IF;
         "Return value (new
                     FitnessInfo(fitness, fitness))"
       END OF METHOD

       METHOD public void Reset()
         "Empty operator"
       END OF METHOD

     END OF REGION

    END OF CLASS
  )

END OF SCHEME
```


DOI 10.15588/1607-3274-2023-1-8

Based on the hyperscheme, SAA schemes for evaluating multiplexers with three, six, and 11 inputs were generated using the IDS toolkit. Further, C# program code for the SharpNEAT framework was generated according to the schemes.

The scheme of the parallel multi-threaded evaluation procedure for the multiplexer example, implemented in SharpNEAT, looks like this:

```
METHOD private void
  Evaluate_Caching(IList<TGenome> genomeList)
  PARALLEL FOR EACH (genome IN genomeList)
  (
    "Get (phenome) for (genome)";
    IF (phenome = null)
    THEN "Decode the (phenome) and store a
              reference against the (genome)"
    END IF;
    IF (phenome = null)
    THEN
      "Set (genome) fitness to (0.0)";
      "Set (genome) auxiliary fitness info to (null)"
    ELSE
      "Evaluate (phenome) and get fitness
        (fitnessInfo)";
      "Set (genome) fitness to (fitnessInfo._fitness)";
      "Set (genome) auxiliary fitness info to
        (fitnessInfo._auxFitnessArr)"
    END IF
  )
END OF METHOD
```

In [5], the distributed version of this procedure was developed for execution on a cloud computing platform.

In this work, the experiments with multithreaded and distributed implementations of neuroevolution of augmenting topology were carried out. A multiplexer with 11 inputs was selected as an example.

The following configurations were chosen as the execution environments for single-process and distributed implementation:

1) local environment, Intel Core i9-9900K CPU (3.60 GHz – 5.00 GHz), 8 cores, 16 logic processors, 32.0 GB RAM, one process, 16 threads;

2) local environment, Intel Core i9-9900K CPU (3.60 GHz – 5.00 GHz), 8 cores, 16 logic processors, 32.0 GB RAM, distributed implementation, 16 local clients-executors;

3) cloud environment, 3rd Gen AMD EPYC Amazon EC2 C6a.large, 3.60 GHz, 2 cores, 4.0 GB RAM, up to 12.5 Gbit/s of network bandwidth, and up to 6600 Mbit/s of storage bandwidth, distributed implementation, 16 local clients-executors;

4) the same cloud environment, but with 32 cloud client executors;

5) the same cloud environment but with 64 cloud client executors.

5 RESULTS

This section gives the results of executing the multiplexer example in the computing environments described above.

Fig. 4 shows the graph of the dependence of the evaluation speed (the number of evaluations per second) on the generation number for local configurations of the environment.

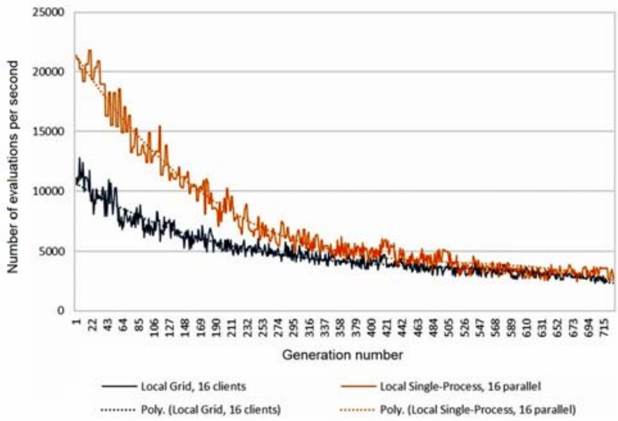

Figure 4 – The graph of the dependence of the evaluation speed on the generation number for local environment configurations

Fig. 5 shows a graph of the evaluation speed on the generation number for the cloud-based environment configurations.

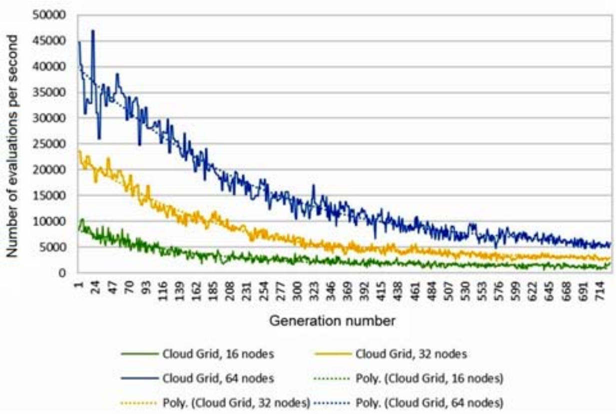

Figure 5 – The graph of the dependence of the evaluation speed on the generation number for cloud environment configurations

6 DISCUSSION

As seen from the graph in Fig. 4, the distributed implementation is expected to show worse results compared to the single-process implementation due to the overhead of interaction between processes. As the complexity of the evaluation task increases (the size of the generated neural network increases), the efficiency of the single-process and local distributed implementation is leveled off, since the overhead costs of computing resources become prohibitively lower than the evaluation costs.

As shown in Fig. 5, the distributed cloud implementation is expected to show worse results (for the

DOI 10.15588/1607-3274-2023-1-8

same number of client-executors) compared to the single-process and local distributed implementation due to the overhead of interaction between the processors of many computers, clients-executors. However, with the growth of the number of executors, we can neglect the constant value of the overhead and obtain a linear increase in the efficiency of the distributed system.

The results of the experiment demonstrated the ability of the distributed system to conduct evaluations on 64 cloud clients-executors and obtain an increase of 60–100% from the maximum capabilities of a single-processor local implementation.

CONCLUSIONS

The scientific novelty of obtained results is that the facilities of hyperscheme algebra are firstly applied for the automated generation of parametric neuroevolution evaluation algorithms on the example of the evaluation problem for a binary multiplexer. A hyperscheme is a high-level parameterized algorithm for solving a certain class of problems. Setting parameter values and subsequent interpretation of the hyperscheme allows obtaining algorithm schemes adapted to specific conditions of their use.

The practical significance of obtained results is that the means of hyperschemes are implemented in the developed integrated toolkit of automated design and synthesis of programs. Based on algorithm schemes, the system generates programs in a target programming language. The advantage of the system is the possibility of describing algorithm schemes in a natural-linguistic form. An experiment was conducted consisting in execution of the generated program for the problem of evaluating a binary multiplexer on a distributed cloud platform, which demonstrated the possibility of the developed distributed system to perform evaluations on 64 cloud clients-executors and obtain an increase in 60–100% of the maximum capabilities of a single-processor local implementation.

Prospects for further research are to apply the algebra-algorithmic method and tools for the automated development of the parallel implementation of evolutionary algorithm evaluation procedure on a graphics processing unit.

ACKNOWLEDGEMENTS

The work is supported by the state budget scientific research project of the Institute of Software Systems of the National Academy of Sciences of Ukraine "Development of methods, technologies and tools for automating parallel programming using methods of computational intelligence" (state registration number 0122U002282).

REFERENCES

1. Doroshenko A. Formal and adaptive methods for automation of parallel programs construction: emerging research and opportunities / A. Doroshenko, O. Yatsenko. – Hershey : IGI Global, 2021. – 279 p. DOI: 10.4018/978-1-5225-9384-3
2. Designing neural networks through neuroevolution / [K. O. Stanley, J. Clune, J. Lehman et al.] // Nature Machine Intelligence. – 2019. – Vol. 1. – P. 24–35. DOI: 10.1038/S42256-018-0006-Z
3. SharpNEAT – Evolution of Neural Networks [Electronic resource]. – Access mode: https://github.com/colgreen/sharpneat
4. BinaryElevenMultiplexerEvaluator [Electronic resource]. – Access mode: https://github.com/colgreen/sharpneat/blob/master/src/SharpNeatDomains/BinaryElevenMultiplexer/BinaryElevenMultiplexerEvaluator.cs
5. Achour I. Z. Distributed implementation of neuroevolution of augmenting topologies method / I. Z. Achour, A. Yu. Doroshenko // Problems in Programming. – 2021. – № 3. – P. 3–15. DOI: 10.15407/pp2021.03.003
6. Yushchenko K. L. Algebraic-grammatical specifications and synthesis of structured program schemas / K. L. Yushchenko, G. O. Tseytlin, A. V. Galushka // Cybernetics and Systems Analysis. – 1989. – Vol. 25, № 6. – P. 713–727. DOI: 10.1007/BF01069770
7. Jiang D. Generation of C++ Code from Isabelle/HOL Specification / D. Jiang, B. Xu // International Journal of Software Engineering and Knowledge Engineering. – 2022. – Vol. 32, № 07. – P. 1043–1069. DOI: 10.1142/S0218194022500401
8. Moreira G. Fully-tested code generation from TLA+ specifications / G. Moreira, C. Vasconcellos, J. Kniess // Systematic and Automated Software Testing : 7th Brazilian Symposium SAST'22, Uberlandia, 3–7 October 2022 : proceedings. – New York : ACM, 2022. – P. 19–28. DOI: 10.1145/3559744.3559747
9. Bonfanti S. Design and validation of a C++ code generator from abstract state machines specifications / S. Bonfanti, A. Gargantini, A. Mashkoor // Journal of Software: Evolution and Process. – 2020. – Vol. 32, № 2. – P. 1–27. DOI: 10.1002/smr.2205
10. Motwani M. Automatically generating precise oracles from structured natural language specifications / M. Motwani, Y. Brun // Software Engineering : 41st IEEE/ACM International Conference ICSE'2019, Montreal, 25–31 May 2019 : proceedings. – Los Alamitos : IEEE, 2019. – P. 188–199. DOI: 10.1109/ICSE.2019.00035
11. Kenneth O. S. A hypercube-based encoding for evolving large-scale neural networks / O. S. Kenneth, D. Ambrosio, J. Gauci // Artificial Life. – 2009. – Vol. 15, № 2. – P. 185–212. DOI: 10.1162/artl.2009.15.2.15202

Received 15.11.2022.
Accepted 11.02.2023.

УДК 004.4'24, 004.896

ПАРАМЕТРИЧНО-КЕРОВАНА ГЕНЕРАЦІЯ ПРОГРАМИ ОЦІНКИ ДЛЯ АЛГОРИТМУ НЕЙРОЕВОЛЮЦІЇ НА ПРИКЛАДІ ДВІЙКОВОГО МУЛЬТИПЛЕКСОРА

Дорошенко А. Ю. – д-р фіз.-мат. наук, професор, завідувач відділу теорії комп'ютерних обчислень Інституту програмних систем НАН України, Київ, Україна.

DOI 10.15588/1607-3274-2023-1-8

Ашур І. З. – аспірант кафедри інформаційних систем та технологій Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського», Київ, Україна.

Яценко О. А. – канд. фіз.-мат. наук, старший науковий співробітник відділу теорії комп'ютерних обчислень Інституту програмних систем НАН України, Київ, Україна.

АНОТАЦІЯ

Актуальність. Розглянуто задачу автоматизованої розробки програм оцінки для алгоритмів нейроеволюції наростаючої топології. Еволюційні алгоритми застосовують механізми мутації, рекомбінації та селекції для пошуку нейронних мереж з поведінкою, яка задовольняє умовам певної формально визначеної задачі. Прикладом такої задачі є знаходження нейронної мережі, що реалізує певну цифрову логіку.

Мета роботи – автоматизоване проектування та генерація програми оцінки для задачі нейроеволюції на прикладі двійкового мультиплексора.

Метод. Методи та інструментальні засоби алгебри алгоритмів Глушкова та алгебри гіперсхем застосовано для параметрично-керованої генерації програми оцінки алгроритму нейроеволюції для бінарного мультиплексора. Алгебра Глушкова покладена в основу алгоритмічної мови, призначеної для багаторівневого структурного проектування та документування послідовних і паралельних алгоритмів та програм у формі, наближеній до природної мови. Гіперсхеми є параметризованими високорівневими специфікаціями, призначеними для вирішення певного класу задач. Задавання значень параметрів і подальша інтерпретація гіперсхем дозволяє отримати алгоритми, адаптовані до конкретних умов їх використання.

Результати. Засоби гіперсхем реалізовано в розробленому інтегрованому інструментарії автоматизованого проектування та синтезу програм. На основі схем алгоритмів система генерує програми цільовою мовою програмування. Перевагою інструментарію є можливість опису схем алгоритмів у природно-лінгвістичній формі. Проведено експеримент з виконання згенерованої програми для задачі оцінки двійкового мультиплексора на розподіленій хмарній платформі. Згадана програма входить до складу SharpNEAT – системи з відкритим кодом, що реалізує алгоритм генетичної нейроеволюції NEAT для платформи .NET. Паралельна розподілена реалізація SharpNEAT була запропонована в попередній роботі авторів.

Висновки. Результати проведених експериментів продемонстрували можливість розробленої розподіленої системи виконувати оцінювання на 64 хмарних клієнтах-виконувачах та отримувати приріст у 60–100 % від максимальних можливостей однопроцесорної локальної реалізації.

КЛЮЧОВІ СЛОВА: алгебра алгоритмів, автоматизоване проектування програм, хмарні обчислення, гіперсхема, нейроеволюція, нейронна мережа, паралельне програмування.

ЛІТЕРАТУРА

1. Doroshenko A. Formal and adaptive methods for automation of parallel programs construction: emerging research and opportunities / A. Doroshenko, O. Yatsenko. – Hershey : IGI Global, 2021. – 279 p. DOI: 10.4018/978-1-5225-9384-3
2. Designing neural networks through neuroevolution / [K. O. Stanley, J. Clune, J. Lehman et al.] // Nature Machine Intelligence. – 2019. – Vol. 1. – P. 24–35. DOI: 10.1038/S42256-018-0006-Z
3. SharpNEAT – Evolution of Neural Networks [Електронний ресурс]. – Режим доступу: https://github.com/colgreen/sharpneat
4. BinaryElevenMultiplexerEvaluator [Електронний ресурс]. – Режим доступу: https://github.com/colgreen/sharpneat/blob/master/src/SharpNeatDomains/BinaryElevenMultiplexer/ BinaryElevenMultiplexerEvaluator.cs
5. Ашур І. З. Розподілена реалізація методу нейроеволюції наростаючої топології / І. З. Ашур, А. Ю. Дорошенко // Проблеми програмування. – 2021. – № 3. – С. 3–15. DOI: 10.15407/pp2021.03.003
6. Yushchenko K. L. Algebraic-grammatical specifications and synthesis of structured program schemas / K. L. Yushchenko, G. O. Tseytlin, A. V. Galushka // Cybernetics and Systems Analysis. – 1989. – Vol. 25, № 6. – P. 713–727. DOI: 10.1007/BF01069770
7. Jiang D. Generation of C++ Code from Isabelle/HOL Specification / D. Jiang, B. Xu // International Journal of Software Engineering and Knowledge Engineering. – 2022. – Vol. 32, № 07. – P. 1043–1069. DOI: 10.1142/S0218194022500401
8. Moreira G. Fully-tested code generation from TLA+ specifications / G. Moreira, C. Vasconcellos, J. Kniess // Systematic and Automated Software Testing : 7th Brazilian Symposium SAST'22, Uberlandia, 3–7 October 2022 : proceedings. – New York : ACM, 2022. – P. 19–28. DOI: 10.1145/3559744.3559747
9. Bonfanti S. Design and validation of a C++ code generator from abstract state machines specifications / S. Bonfanti, A. Gargantini, A. Mashkoor // Journal of Software: Evolution and Process. – 2020. – Vol. 32, № 2. – P. 1–27. DOI: 10.1002/smr.2205
10. Motwani M. Automatically generating precise oracles from structured natural language specifications / M. Motwani, Y. Brun // Software Engineering : 41st IEEE/ACM International Conference ICSE'2019, Montreal, 25–31 May 2019 : proceedings. – Los Alamitos : IEEE, 2019. – P. 188–199. DOI: 10.1109/ICSE.2019.00035
11. Kenneth O. S. A hypercube-based encoding for evolving large-scale neural networks / O. S. Kenneth, D. Ambrosio, J. Gauci // Artificial Life. – 2009. – Vol. 15, № 2. – P. 185–212. DOI: 10.1162/artl.2009.15.2.15202

DOI 10.15588/1607-3274-2023-1-8