Control and Cybernetics

vol. **39** (2010) No. 4

Searching for optimal size neural networks in Assembler Encoding*

by

Tomasz Praczyk

Polish Naval Academy, Institute of Naval Weapon
Gdynia, Poland
e-mail: t.praczyk@amw.gdynia.pl

Abstract: Assembler Encoding represents a neural network in the form of a simple program called Assembler Encoding Program. The task of the program is to create the so-called Network Definition Matrix, which maintains all the information necessary to construct a network. To generate the programs and, in consequence, neural networks, evolutionary techniques are used.

One of the problems in Assembler Encoding is to determine an optimal number of neurons in a neural network. To deal with this problem a current version of Assembler Encoding uses a solution that is time consuming and hence rather impractical. The paper proposes four other solutions to the problem mentioned. To test them, experiments in a predator-prey problem were carried out. The results of the experiments are included at the end of the paper.

Keywords: evolutionary neural networks.

1. Introduction

Using evolutionary techniques to generate Artificial Neural Networks (ANNs) is usually connected with encoding the latter. There are many ANN encoding methods, e.g. connectivity matrix (CM) proposed by Miller, Todd and Hedge (1989), Kitano's matrix rewriting encoding scheme (Kitano, 1990), Gruau's cellular encoding (Gruau, 1994, 1995; Gruau, Whitley and Pyeatt, 1996), edge encoding proposed by Luke and Spector (1996), Symbiotic Adaptive NeuroEvolution (SANE) devised by Moriarty and Miikkulainen (Moriarty, 1997; Moriarty and Miikkulainen, 1998), or the schemes proposed by Nolfi and Parisi (1992) and Cangelosi, Parisi and Nolfi (1994). One of the encoding methods is Assembler Encoding (AE) (Praczyk, 2007a,b; 2008). AE originates from the cellular and edge encoding but it also has features common with Linear Genetic Programming, presented, in particular, in Krawiec and Bhanu (2005) and Nordin,

*Submitted: February 2009; Accepted: April 2010.

Banzhaf and Francone (1999). AE assumes that ANN is represented in the form of a program (Assembler Encoding Program - AEP) whose structure is similar to the structure of a simple assembler program. The task of AEP is to create the Network Definition Matrix (NDM) containing all the information necessary to produce ANN. The process of ANN construction consists of three stages. First, Genetic Algorithm (GA) is used to produce AEPs. Then, each AEP creates and fills up NDM. Once the matrix is created, it is transformed into ANN.

To completely define ANN, it is necessary to determine the following: the number of neurons, the type of each neuron, connectivity, values of parameters of each neuron and connection. In AE, all the information mentioned above, except for the number of neurons, is included in components of NDM. The number of neurons is dependent on the size of NDM. In the current version of AE, NDMs and, in consequence, ANNs grow gradually. Initially, all AEPs operate on NDMs containing a minimal acceptable number of rows and columns. This means that all ANNs created at the beginning of the evolution consist only of input and output neurons. Afterwards, if all AEPs created over an assumed period are not able to generate any satisfactory solution, NDMs are expanded by one row and one column, which corresponds to expansion of ANNs by one neuron. The process of growth of NDMs continues until the moment when all of them represent ANNs of a maximal acceptable number of neurons.

At a first glance it is apparent that such procedure can only be useful when creating small ANNs. Generating larger neural architectures according to the procedure described above would rather be a long-lasting process. The paper proposes other methods that make it possible to determine the final number of neurons in ANNs. The first method assumes that all AEPs act on NDMs of a maximal size. In the second method, information about the number of neurons is located in a chromosome. The third solution uses AEPs that have a potential to expand NDMs on which they act; initially all NDMs are of a minimal size. The last, fourth method is a combination of the second and the third ones. First, NDM is constructed of the size stored in a chromosome. Then, AEP can change the size of the matrix. To this end, it uses operations that add or remove columns and rows from NDM.

All the methods above were tested in a simple version of the predator-prey problem. In the experiments, the task of each ANN was to control three agents-predators. The common goal of the predators was to capture an escaping agent-prey behaving according to a simple deterministic strategy. Since the speed of each predator was lower than or equal to the speed of the prey, the predators had to cooperate to accomplish the goal.

The paper is organized as follows: Section 2 is a short presentation of AE; Section 3 is a detailed description of all techniques that can serve to determine the number of neurons in ANNs; Section 4 is an illustration of experimental results; Section 5 is the summary.

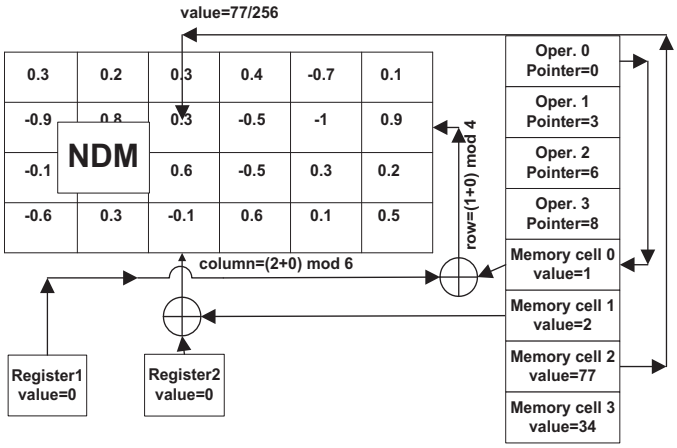


Figure 1. Diagram of AE (AEP presented on the right includes four operations and four memory cells. Operation 0 changes a single element of NDM. To this end, it uses three consecutive memory cells. The first two cells store an address to the element of NDM being updated. To determine a final address of the element mentioned, values of registers are also used. The third memory cell used by the Operation 0 stores a new value of the element. The value is scaled before NDM is updated. A pointer to the memory part of AEP where three cells used by Operation 0 are located is included in Operation itself.)

2. Fundamentals of Assembler Encoding

In AE, an ANN is represented in the form of a program called Assembler Encoding Program. AEP is composed of two parts, i.e. the part including operations (the code part of AEP) and the part including data (the memory part of AEP). The task of AEP is to create NDM and to fill it in with values. To this end, AEP uses the operations. The operations are run one after another. When working, the operations use data located at the end of AEP (Fig. 1). Once the last operation finishes its work, the process of creating NDM is completed. The matrix is then transformed into ANN (Fig. 2).

AEPs can use various operations. The main task of most operations is to modify NDM. The modification can involve a single element of the matrix or a group of elements. Fig. 3 shows the implementation of two example operations.

The task of CHG is to change a single element in NDM. The new value of the element, stored in parameter p_0, is scaled to $< -1, 1 >$. An address of the element being changed depends on both parameters p_1, p_2 and registers R_1, R_2. A role of the registers is detailed in the further part of the paper.

CHGC0 modifies elements of NDM located in a column indicated by p_0 and R_2. The number of elements being updated is stored in p_2. An index of the first

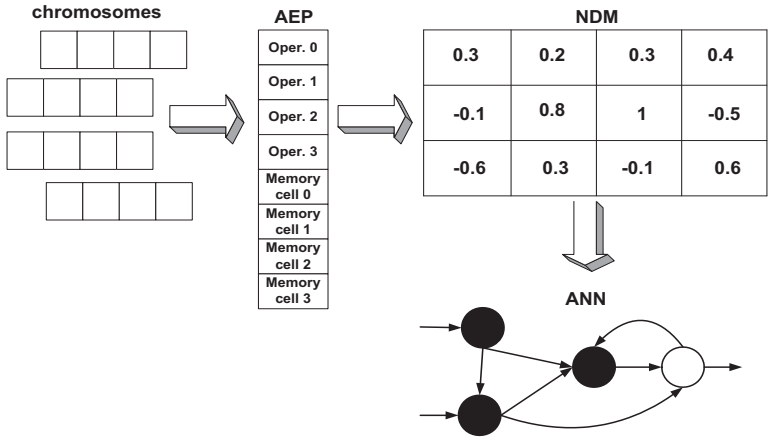


Figure 2. Creating ANN in AE

```
CHG(p0,p1,p2,*)
{
row=(abs(p1)+R1)mod NDM.width;
column=(abs(p2)+R2)mod NDM.height;
NDM[row,column]=p0/MaxValue;
}
```

```
CHGC0(p0,p1,p2,*)
{
column=(abs(p0)+R2)mod NDM.height;
numberOfIterations=abs(p2)mod NDM.width;
for(i=0;i<=numberOfIterations;i++){
    row=(i+R1)mod NDM.width;
    NDM[row,column]=D[(abs(p1)+i)×
      mod D.length]/MaxValue;}
}
```

Figure 3. CHG and CHGC0 operations (Parameters unimportant for implementation of operations are marked by *, $D[i]$ denotes i^{th} data in AEP)

element being updated is located in R_1. To update elements of NDM CHGC0 uses data from AEP. An index to a memory cell including the first data used by CHGC0 is stored in p_1.

In addition to operations, whose task is to modify the content of NDM, AE also uses a jump operation denoted as JMP. The jump makes it possible to repeatedly use the same code of AEP in different places of NDM. This is possible owing to the change of values of the registers once the jump is carried out. An example use of the jump is demonstrated in Fig. 4. The program presented in the figure proceeds as follows. First, both registers are initiated to 0. Then, the first two operations are carried out, the result of which is visible in the top left corner of NDM. In the next step, the jump denoted in the figure as JMP(0,2,0,*) is run. It first updates the values of the registers and then control goes back to the first operation of AEP. R_1 is set to 0 (Memory cell 0) whereas R_2 to 2 (Memory cell 1). At this point, the two operations preceding the jump are carried out once again. This time, however, both operations update a different

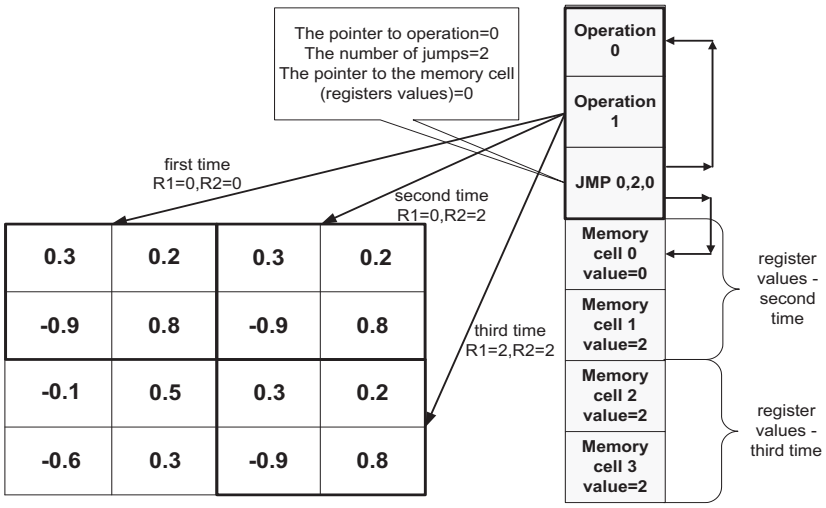
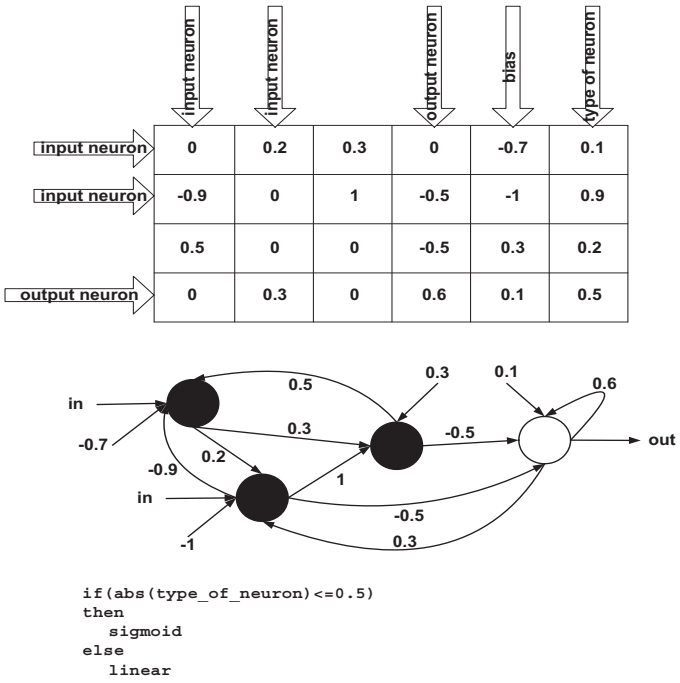


Figure 4. JMP operation




```
if(abs(type_of_neuron)<=0.5)
then
   sigmoid
else
   linear
```

Figure 5. NDM as connectivity matrix

fragment of NDM. Since the jump is run twice, each time with different values of the registers, the two first operations of AEP are executed in three different areas of NDM.

Once AEP finishes its work, the process of transforming NDM into ANN is started. To make it possible to construct ANN based on NDM, the latter has to include all the information necessary to create the network. When we wish to create the same skeleton of ANN, i.e. ANN without determined weights of interneuron connections, NDM can take the form of the classical CM, i.e. a square binary matrix with the number of rows and columns equal to the number of neurons. The value "1" in i^{th} column and j^{th} row of such a matrix means a connection between i^{th} neuron and j^{th} neuron. In turn, the value "0" denotes lack of connection between these neurons. When the purpose is to create complete ANN with determined values of weights, types of neurons, and parameters of neurons, NDM should take the form of a real valued variety of CM with extra columns or rows containing definitions of individual neurons. The example of such a matrix is presented in Fig. 5.

3. Evolution in Assembler Encoding

In AE, evolution of AEPs proceeds according to a scheme proposed by Potter and De Jong (Potter, 1997; Potter and De Jong, 2000). The scheme assumes a division of a solution created in an evolutionary manner into parts. Each part evolves in a separate population. The complete solution is formed from selected representatives of each population. In order to use the scheme above in relation to AEPs, it is necessary to divide them into parts. In the case of AEPs, the division is natural. The operations and data make up natural parts of AEPs. Since the evolutionary scheme chosen assumes the evolution of each part in a separate population, AEP consisting of n operations and a sequence of data evolves in n populations with operations and one population with data (Fig. 6). During the evolution, AEPs expand gradually. Initially, all AEPs include one operation and a sequence of data. The operations and data come from two different populations. When the evolution stagnates, i.e. lack of progress in fitness of generated solutions is observed over some period, a set of populations containing the operations is enlarged by one population. This procedure extends all AEPs by one operation. Each population can also be replaced with a newly created population. Such situation takes place when the influence of all individuals from a given population on fitness of generated solutions is definitely lower than the influence of individuals from the remaining populations (a population can be replaced when, for example, fitness of a population, measured as the average fitness of all individuals from the population, is definitely lower than the fitness of the remaining populations).

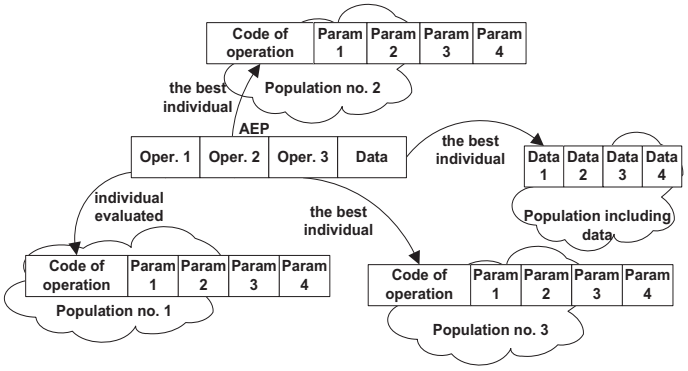


Figure 6. Evolution of AEPs for n=3

4. Encoding size of ANN in AE

In order to completely define ANN, it is necessary to determine the following: the number of neurons, topology of interneuron connections, types of neurons, parameters of neurons and connections. In AE, an exhaustive definition of ANN is included in NDM. The size of the matrix determines a maximal number of neurons whereas elements of NDM define the remaining parameters of ANN. NDM is created by AEP. In order to initiate NDM, AEP has to contain information about the size of the matrix. In the current version of AE, NDMs grow gradually (Fig. 7). Initially, all AEPs act on NDMs of a minimal size. Such matrices correspond to ANNs including exclusively input and output neurons. Further, if no AEP can generate satisfactory ANN over a longer period, all NDMs are augmented by one row and one column. Such a procedure corresponds to augmenting all ANNs by one hidden neuron. Expanding NDMs and thereby ANNs continues to the point when ANN that meets our expectations is generated.

The scheme of creating ANNs described above can be successfully applied in the situation when ANNs with a small number of hidden neurons are needed. Using the scheme above to construct more complex neural architectures is rather impractical. To generate ANNs consisting of m hidden neurons, AEPs first have to form networks consisting of one hidden neuron, two hidden neurons, three hidden neurons, and so on. ANNs consisting of m hidden neurons are created after m long steps. For large m this process may last definitely too long.

The paper presents four alternative schemes for creating ANNs. Each scheme makes it possible to create ANNs with a different number of neurons from the very beginning of the evolutionary process. All the schemes assume that each AEP, regardless of the point of creating, has a potential to generate ANN unique in terms of connectivity, weights of interneuron connections, types of neurons as well as in terms of the number of neurons.

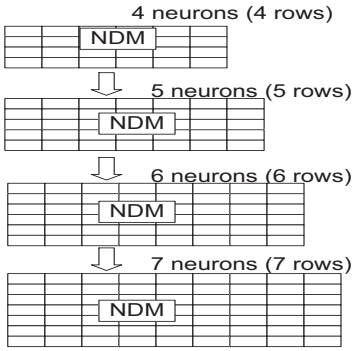


Figure 7. Gradual enlargement of NDMs used in current version of AE

4.1. Method no. 1

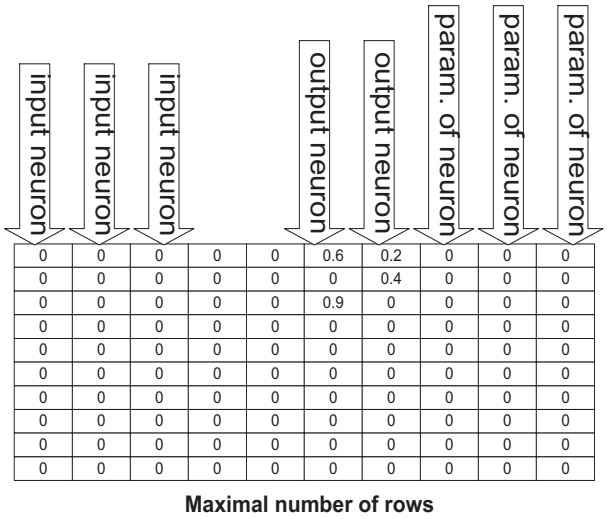


Figure 8. Illustration of method no. 1

In method no. 1, all AEPs operate on NDMs of a maximal acceptable number of rows and columns (size of NDM remains constant throughout work of AEP; Fig. 8). This means that each AEP has a potential to create ANN of a maximal number of neurons. However, such a situation takes place only if all neurons considered in NDM have a connection (direct or indirect) with output neurons. Once some neurons are separated from the output, ANN containing fewer neurons is created. Neurons isolated from the output have no influence on behavior of ANN and they can be removed from the network.

In method no. 1, only AEP decides about the size of ANN. If it fills in most rows and columns in NDM, ANN arises containing a larger number of neurons. In turn, if working of AEP involves few cells of NDM, ANN is created containing few neurons (Fig. 9).

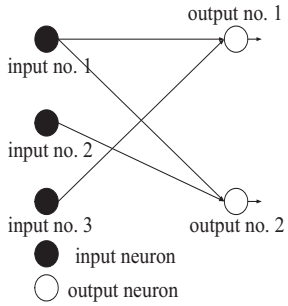


Figure 9. ANN whose NDM is presented in Figure 8

4.2. Method no. 2

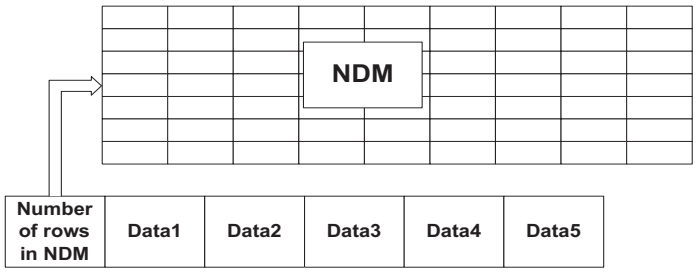


Figure 10. Illustration of method no. 2

In this case, the information concerning the size of NDM is located in a chromosome with data (Fig. 10). Typically, this chromosome exclusively includes data for AEP. In the method considered, the information about the size of NDM is added to data. AEP is formed from a set of chromosomes-operations and from one chromosome-data. Once a process of integrating AEP is completed, the program initiates NDM. The size of NDM is located in the first part of the chromosome-data. It is assumed that the number of rows in NDM cannot be lower than the total number of input and output neurons in ANN. As in the previous case, the size of NDM remains constant throughout the working of AEP. Once AEP stops working, ANN is created. All neurons isolated from the output of the network are then removed from it. In method no. 1, each AEP has a potential to create ANN of a maximal acceptable number of neurons.

Method no. 2 enables AEP to create ANN, whose maximal size is encoded in a chromosome with data.

4.3. Method no. 3

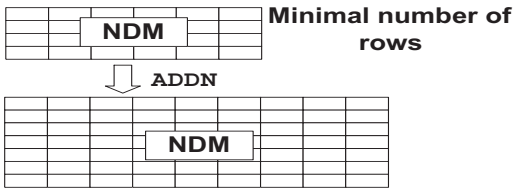


Figure 11. Illustration of method no. 3

This time, AEPs have a potential to expand NDMs during their work. Initially, each AEP initiates NDM of the size corresponding to ANN containing exclusively input and output neurons. Then, AEP can use ADDN operation to expand NDM by some number of rows and columns (Fig. 11). Adding new rows and columns to NDM corresponds to adding new neurons to ANN – neurons unconnected with the rest of ANN. The number of neurons being added is a parameter of ADDN. Addition of new neurons does not destroy connections already established in ANN. As before, all neurons separated from the output of ANN are removed from it.

4.4. Method no. 4

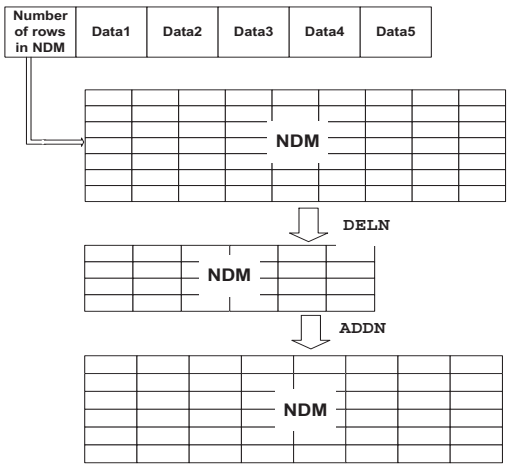


Figure 12. Illustration of method no. 4

This method is a combination of methods no. 2 and 3. The initial size of NDM is encoded in a chromosome with data. Then, AEP has a potential to modify the size of NDM through using operations ADDN and DELN (Fig. 12; an example of AEP created in the experiments with method no. 4 is presented in Fig. 16). The operation ADDN adds new neurons to ANN. It does not destroy connections that already exist in the network. The task of DELN is to remove a single neuron from ANN. The number of the neuron is a parameter of the operation. Elimination of the neuron practically takes place through removing corresponding row and column from NDM. As before, all neurons separated from the output of ANN are removed from it.

5. Experiments

In order to compare the methods presented above, experiments on the predator-prey problem were carried out. During the tests, the task of ANNs was to control a set of cooperating predators whose common goal was to capture a fast moving prey. The experiments were performed in a configuration with one prey and three chasing predators. The prey behaved according to a simple algorithm, whereas the predators were controlled by a single ANN.

5.1. Environment

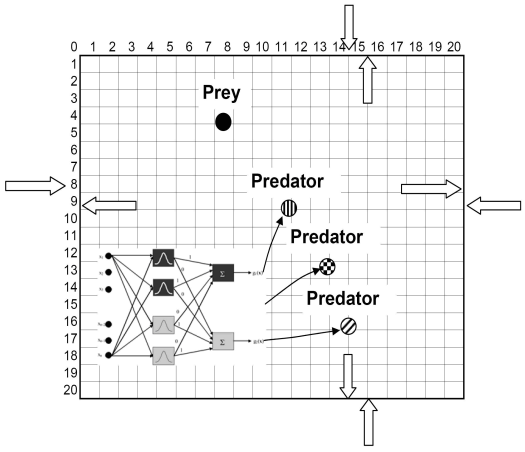


Figure 13. Artificial world in which the task of predators is to capture prey

The predators and the prey lived in a common environment. We used 20x20 square without obstacles to represent the environment (Fig. 13). In order to ensure infinite space for the predators and prey and for their struggle, we made the environment open at each side. This means that every attempt to move beyond the upper, lower, right or left border of the square caused the object

making such an attempt to move to the opposite side of the environment. As a result, a simple strategy of predators consisting in chasing the prey did not work. In such a situation, the prey in order to evade predators, could simply escape up, down, right or left.

5.2. Residents of artificial world

In our experiments, three predators and one prey coexisted in the artificial environment. The predators were controlled by ANN produced by AEP. They could select five actions: to move in North, South, West, East direction or to stand still. The length of step made by every predator was 1 while the step made by the prey was equal either 2 or 1. In order to capture the prey, the predators had to cooperate. Their speed was either two times lower or the same as speed of the escaping prey so they could not simply chase the prey to grasp it. We assumed that the prey was captured if the distance between it and the nearest predator was lower than 2.

In the experiments, we used two types of prey, i.e. a simple prey and an advanced prey. With regard to the simple prey, it was controlled by a simple algorithm which forced it to move directly away from the nearest predator but solely in the situation when distance between it and the nearest predator was lower or equal 5. In the remaining cases, i.e. when no predator was closer to the prey than the assumed distance, the prey did not move. The prey, when running away, could select four actions: to move in North, South, West or East direction. The strategy of the simple prey is presented below:

$$\pi_{simple}(s) = \begin{cases} \text{StandStill} \quad \text{if} \quad d(p,s) > 5 \quad \forall p \in P \\ arg\max\limits_{a \in A} D\Big(s, a, arg\min\limits_{p \in P} d(p,s)\Big) \quad \text{otherwise} \end{cases} \tag{1}$$

where
P - set of predators,
A - set of actions of prey ($A = \{\text{StandStill, North, South, West, East}\}$),
$d(p,s)$ - distance between prey and predator p in state of environment s,
$D(s,a,p)$ - distance between prey and predator p in the state of environment, which is the direct consequence of action a performed by prey in state s.

Making decision, the advanced prey, unlike its simpler counterpart, always took into consideration the location of all predators situated close to it. Actions performed by the advanced prey always maximized the average distance between the prey and all predators that were close to it. Other aspects of behavior of the advanced prey, i.e. behavior away from the predators and actions which the prey could perform in each step, were the same as in the case of the simple prey.

The strategy of the advanced prey is presented below:

$$\pi_{advanced}(s) = \begin{cases} \text{StandStill} \quad \text{if} \quad d(p,s) > 5 \quad \forall p \in P \\ arg\max\limits_{a \in A} \left(\frac{1}{|P_5(s)|} \sum_{p \in P_5(s)} D(s,a,p) \right) \quad \text{otherwise} \end{cases} \tag{2}$$

where
$P_5(s) = \{p \in P,\ d(p,s) \leq 5\}$.

5.3. Neural controllers

NDMs generated during the experiments usually represented recurrent ANNs. However, we decided that controllers used in the experiments should have a feed-forward architecture. In order to obtain such ANNs, we used only elements of NDMs localized in their upper parts. The remaining elements were neglected during the process of ANN construction.

ANNs contained three types of neurons: radial, sigmoid and linear neurons. Information about the type of neuron was located in an additional column of NDM. Each matrix included three additional columns. The remaining two columns contained information about bias and value of one parameter of each neuron.

ANNs had usually six inputs and three outputs (in some cases ANNs did not require so many inputs to effectively control the predators). The number of outputs corresponded to the number of predators. In turn, the number of inputs was twice the number of predators. Each output gave commands to one predator. In turn, each input informed about vertical or horizontal distance between the prey and one of the predators.

5.4. Parameters of evolutionary process

In all the experiments, the evolution of operations and data proceeded according to the canonical GA. All chromosomes used in the experiments consisted of 7-bit blocks of genes. Every chromosome-operation consisted of five blocks of binary genes (one block for code of an operation and the remaining four blocks for parameters of the operation). The list of applied operations is presented at the end of the paper (see Appendix 1). Chromosomes-data could change the length during consecutive co-evolutionary cycles. In order to make such a change possible, GA that processed a population with data, in addition to crossover and mutation, used a cut-splice operator (Fig. 14). The implementation of the crossover always produced an offspring of the same length as parents. The cut-splice operator which was always activated after the crossover and mutation modified the size of a chromosome through addition or removal of a single block of genes (single data) from the same end of the chromosome. In the experiments, we assumed that the chromosomes-data can maximally contain 20 data, i.e. 20 7-bit blocks of genes. Each use of an excessive number of data caused drastic

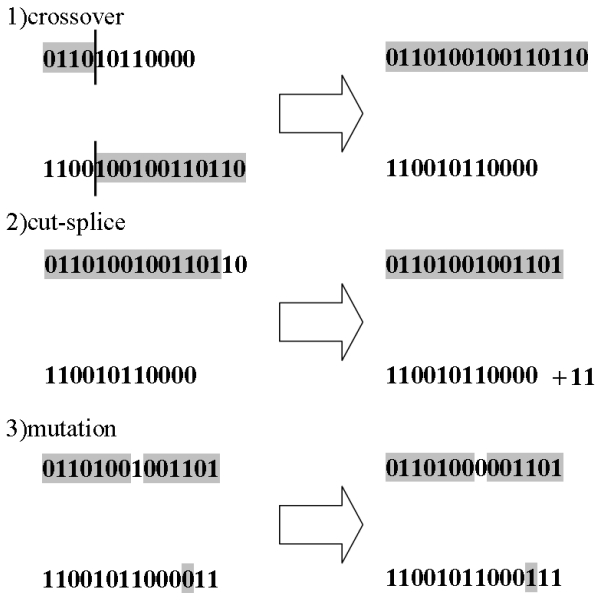


Figure 14. Evolutionary operators used in experiments: one-point crossover (chromosomes could be cut in two only between two separate blocks of binary genes, e.g. between two parameters of operation or between two integers in data), per-bit mutation, and cut-splice

decrease in fitness of AEP. In our experiments, we assumed a maximal number of operations which could be possessed by each AEP: 12 operations. Initially, every AEP contained one operation and one set of data from two different populations. Consecutive populations with operations were added after every 5000 co-evolutionary cycles if the generated AEPs were not able to achieve progress in performance within this period. Populations with operations and data could also be replaced by newly created populations when the contribution of a substituted population to created AEPs was considerably less than the contribution of the remaining populations. The contribution of a population was measured as average fitness of individuals belonging to that population. The remaining values essential for the experiments are presented below:

- population size: 20 individuals (operations and data),
- maximum number of co-evolutionary cycles: 200 000 (In the case of the classic variant of AE, we dealt with four separate evolutionary attempts. For the first 50 000 cycles, AEPs operated on NDMs of the minimal acceptable size 9 input and output neurons. When they could not generate any successful ANN[1] within this period, all NDMs were augmented by one

[1]successful ANN is an ANN which resulted in capturing the prey in all testing scenarios

column and row and a next attempt consisting of 50 000 cycles started. This process was maximally repeated four times, i.e. over 200 000 cycles. In this way, the classic variant of AE could produce ANNs with the maximum number of neurons equal to 13, i.e. 4 hidden neurons and 9 input and output neurons. In the case of the methods discussed in the paper, evolutionary settings did not change over all 200 000 cycles),

- maximum size of NDMs (except for the classic variant of AE): 30 rows and 33 columns (which means 9 input and output neurons, and 21 hidden neurons),
- crossover probability: 0.7 (operations and data),
- per-bit mutation probability: 0.05 (operations), 0.01 (data),
- chromosome extension probability: 0.1 (exclusively in chromosomes-data).

5.5. Evaluation process

In order to evaluate ANNs, we used twenty different scenarios. The tests proceeded in the following way. At first, each ANN was tested in the scenario no. 1. If the predators could not capture the prey during some assumed period, the test was stopped and ANN was assigned appropriate evaluation, depending on the distance between the prey and the nearest predator. However, if the predators grasped the prey, they were put to a test according to a next scenario. During the experiments, we assumed that the predators could perform 100 steps before a scenario was interrupted.

The scenarios differed in the initial position of the prey, in the length of step of the prey and in the type of the prey applied (simple or advanced). Consecutive scenarios were more and more difficult. At first, the predators had to capture the simple prey that was as fast as them. The predators, having passed the first exam, had to pit against the simple prey that was twice faster than the predators. In the next step, the speed of the prey was decreased once again. However, this time the predators had to face the advanced prey which took better decisions than its predecessor. In the last stage, the predators which coped with all earlier scenarios had to capture the advanced, fast prey. In all the scenarios, starting positions of all three predators were the same (Fig. 15): they always started from position (0,0). All of the twenty scenarios are specified below:

- Scenario no 1,6,11,16: starting position of prey (5,5) – position 1,
- Scenario no 2,7,12,17: starting position of prey (15,5) – position 2,
- Scenario no 3,8,13,18: starting position of prey (5,15) – position 3,
- Scenario no 4,9,14,19: starting position of prey (15,15) – position 4,
- Scenario no 5,10,15,20: starting position of prey (10,10) – position 5,
- Scenario no 1-5: simple prey, prey step = 1,
- Scenario no 6-10: simple prey, prey step = 2,
- Scenario no 11-15: advanced prey, prey step = 1,

- Scenario no 16-20: advanced prey, prey step = 2 (example behavior of predators and prey in scenarios no. 18 and 20 is presented in Fig. 17).

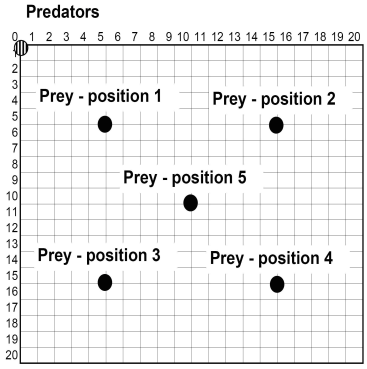


Figure 15. Initial locations of prey and predators

a)

```
CHGM1|59|19|58|34
CHGM2|45|-38|45|2
ADDN|47
CHGM2|30|29|-16|43
Data:47|18|-39|28|58|-57|-52|8|31|
-51|30|45|-45|-18|-1|-18|-54|58|5|
-59|16|36|-32|19|-19
```

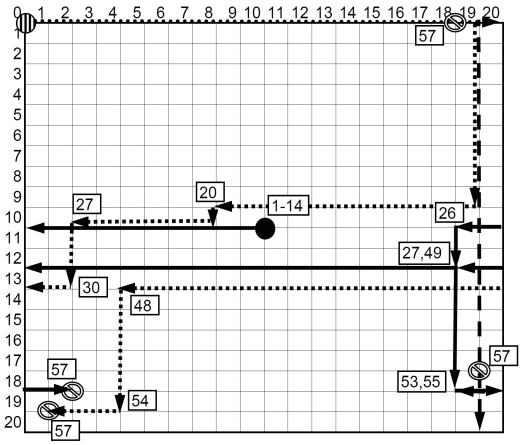
b)

```
Operations:
1010011 0110111 0110010 0010111 0010001
0100001 0101101 1011001 0101101 0010000
0100010 0111101 1011100 0001100 1010001
0111100 0011110 0101110 1000010 0110101
Data:
0111101 0010010 1111001 0001110 0010111
1100111 1001011 0000100 0111110 1110011
0011110 0101101 1101101 1010010 1100000
1010010 1011011 0010111 0101000 1110111
0000010 0001001 1000001 0110010 1110010
```

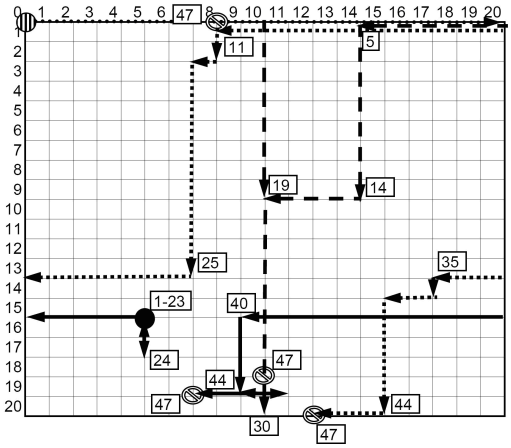
c)

```
 0.4  0.9   0     0     0    0   0 0 0 0 0  0   -0.2 -0.5  -0.8 -0.2  -0.3
-0.8  0     0     0     0    0   0 0 0 0 0  0    0.03 0.3   0.1 -0.8   0.7
 0.4  0    -0.3  -0.4  -0.4  0.1 0 0 0 0 0 -0.4 -0.5 -0.6   0.4  0.9   0.2
 0.7  0     0.2   0.2   0.4 -0.1 0 0 0 0 0 -0.4 -0.5  0.5  -0.8  0.07 -0.6
-0.7  0     0.4   0.3   0.03 0.3 0 0 0 0 0  0.06 0.4  0.4   0.4 -0.9   0.4
 0.9 -0.2  -0.4  -0.3  -0.4  0.1 0 0 0 0 0  0.2 -0.5  0.07 -0.6  0.7   0.2
 0    0     0     0     0    0   0 0 0 0 0  0    0    0     0    0     0
 0    0     0     0     0    0   0 0 0 0 0  0    0    0     0    0     0
 0    0     0     0     0    0   0 0 0 0 0  0    0    0     0    0     0
 0    0     0     0     0    0   0 0 0 0 0  0    0    0     0    0     0
 0    0     0     0     0    0   0 0 0 0 0  0    0    0     0    0     0
-0.9 -0.01 -0.4  -0.1   0.1 -0.3 0 0 0 0 0 -0.4 -0.1 -0.9   0.4 -0.7   0.5
-0.8 -0.2   0.06 -0.007 0.2  0.2 0 0 0 0 0  0.2  0.8  0.2   0.9 -0.2  -0.5
 0.1 -0.8   0.2  -0.1  -0.2  0.4 0 0 0 0 0  0.3 -0.3  0.5  -0.9 -0.01  0.3
```

Figure 16. (a) Example of AEP, which created successful ANN – method no. 4, (b) encoded form of AEP presented in point (a), (c) NDM generated by AEP presented in points (a) and (b)



(a)



(b)

Figure 17. Example of behavior of predators and prey in scenario no. 20 (a) and scenario no. 18 (b) (neuro-controller: ANN whose NDM is presented in Fig. 16c). Circles indicate initial positions of predators and prey (black circle prey, circle with vertical stripes predators), round symbols with diagonal lines denote final positions, arrowed lines indicate directions of movement (solid line prey, dashed or dotted lines predators), whereas black boxes show the time of appearance of individuals in a given place

To evaluate ANNs, the following fitness function was used:

$$f(ANN) = \sum_{i=0}^{n} f_i \tag{3}$$

$$f_i = \begin{cases} d_{max} - \min\limits_{p} d_i(p), \text{ prey not captured in } i^{th} \text{ scenario} \\ f_{captured} + (100 - m_i)/a, \text{ prey captured in } i^{th} \text{ scenario} \\ 0, \text{ prey not captured in previous scenario} \end{cases} \tag{4}$$

where
f_i - reward received in i^{th} scenario
$d_i(p)$ - distance between prey and predator p in the end state of i^{th} scenario
d_{max} - maximum distance between two points in the environment
$f_{captured}$ - reward for grasping prey in a single scenario ($f_{captured} = 100$)
m_i - number of steps to capture prey ($m_i < 100$)
a - this value prevented the situation, in which partial success was better than success in all scenarios
n - number of scenarios ($n = 20$).

The total fitness value of each ANN is a sum of rewards from scenarios in which ANN had taken part. The reward for a scenario depends on a chase result. In the case of success, ANN obtains extra fitness for grasping the prey and additionally a reward reversely proportional to the number of steps which the predators had to make to capture the prey. In the case of failure, ANN obtains fitness proportional to the distance between the prey and the nearest predator.

5.6. Experimental results

To compare individual methods proposed in the paper, their learning abilities, i.e. abilities to produce ANNs effective in learning tasks, and speed were measured. ANNs generated during the experiments were not tested in terms of generalization performance. An ability of ANNs to generalize knowledge is a quality indicator of the whole method but not of individual modifications. The shape of ANNs produced in AE depends on operations used by AEPs. They form NDMs and thereby ANNs. In the experiments, all the methods could produce AEPs equipped with the same types of operations (except for operations whose task is to change the size of NDMs; the influence of such operations is, however, restricted to adding or removing neurons separated from the rest of ANN, they do not determine a structure of connections in ANNs, which mainly influences the behavior of the networks) and there is no reason to think that each of them could prefer different operations and thereby produce ANNs with completely different characteristics. A feature distinguishing individual methods is their speed in creating effective ANNs but not ANNs themselves. For that reason, the generalization tests were not carried out.

Table 1. Results of experiments (1 - Average fitness of ANNs, 2 - % of successful evolutionary attempts, i.e. attempts in which successful ANN was produced, 3 - Average number of co-evolutionary cycles necessary to generate successful ANN)

	Method				Gradual
	no. 1	no. 2	no. 3	no. 4	growth method
1	1988.5	2059.1	1996.3	2062.8	2073.2
2	84%	100%	92%	100%	100%
3	135428.5	93881.2	142085.7	111629.8	90886.2

In the experiments, whose results are included in Table 1, all the methods presented in Section 4 were tested fifty times (each method was represented by fifty ANNs, i.e. fifty evolutionary attempts were performed for each method). For the comparison purposes, the method based on gradual growth of NDMs was also tested. During the tests, it turned out that all the methods proposed in the paper are able to prepare effective ANNs within the assumed learning time (200,000 evolutionary generations). The only incompletely successful methods (not all evolutionary attempts succeeded, i.e. ended in producing a successful ANN) were methods no. 1 and no. 3. In the former case, the successful ANNs were produced in 84% of evolutionary attempts, whereas in the latter one, the result was somewhat better and it amounted to 92% of complete successes. In the remaining cases, all the attempts were successful. As for the average fitness of ANNs, results are similar to the ones presented above. Again, methods no. 1 and no. 3 turned out to be less effective than the remaining solutions. In both cases, values close to 2000 were attained, meaning that ANNs were successful in 19 scenarios, on average. Values above 2000 obtained in the remaining cases indicate the complete success in all the fifty evolutionary attempts. With regard to the speed in generating effective ANNs, the experiments showed that the methods proposed in the paper are in most cases slower than the one used in the classical variant of AE. The only exception is method no. 2, which was almost as fast as the gradual growth of ANNs. It seems that the main reason of such a state of affairs is the fact that most methods proposed in the paper (except for method no. 1) enable AEPs to operate on NDMs of a varied size during all the evolutionary process. In this way, their AEPs have a more difficult task than the counterparts from the gradual growth method. AEPs, which are effective for NDMs of some fixed size, are completely useless for NDMs of a different size. In the gradual growth method, the objective of all AEPs is only to fill in NDMs with values. Since all NDMs are of equal size, there is no need for the evolution and AEPs to address this problem. Method no. 1, like the gradual growth method, does not change the size of NDMs during the evolution. However, this method usually uses NDMs of a larger size than NDMs from the remaining methods. The size of NDMs seems to be the main reason of worse results of the

method. Determining connectivity in ANN including, say, 30 neurons is simply much more difficult than doing so in ANN with, say, 10 neurons.

In sum, the experiments showed that the methods proposed in the paper are not as fast as the gradual growth of NDMs and they rather should not be used to create small ANNs and ANNs whose number of neurons is known or roughly known in advance. However, in most real problems, the size of ANN is not known beforehand. In such a case, the methods proposed in the paper can be used. All of them can be applied either to determine a rough size of ANN or to find a final solution. Using the method based on the gradual growth of NDM, in this situation, would probably prolong the time necessary to generate a satisfactory solution.

However, an interesting idea seems to be to combine the gradual growth method with the methods proposed in the paper. The latter methods, as mentioned above, could be used to determine a rough size of ANN. Then, the gradual growth method would be activated. Its task would be to find the final form of ANN. To test this idea, simple experiments were carried out. In these experiments, all the proposed methods, except for method no. 1, were first used to generate incompletely effective ANNs, i.e. ANNs of fitness greater than 1500 (the prey captured in at least 15 scenarios). In the next step, responsibility for generating completely successful ANNs was shifted onto the gradual growth method. Results of the experiments are summarized in Table 2. Since in all the examined cases we achieved 100% of effectiveness, the table only compares the speed of the solutions tested. Generally, it appeared that the combination of the methods yields positive results. In two cases out of the three, the improvement in the speed in relation to the previous experiments was noted. Noteworthy is the fact that in all the tests the size of NDMs, suggested by the methods proposed in the paper, was sufficient to generate a successful ANN. In no case, the gradual growth method had to enlarge NDMs and to start the evolution from the beginning. Another issue worth mentioning is the degree of improvement in the case of method no. 2. It seems that the main reason why such result was accomplished is a way of combining method no. 2 and the gradual growth method. In this case, all operations and data prepared by method no. 2 were directly transferred further to a next evolutionary stage in which the gradual growth method was used. Thus, the evolution did not start from a random point but from a place indicated by method no. 2. Both methods use the same set of operations so there was no obstacle to combine them in such a way. To apply such a solution, it was only enough to omit information included in chromosomes-data about size of NDMs. For the remaining methods, the approach above could not be used and the gradual growth method always had to start from randomly generated operations and data. The main impediment, in this case, was the use of operations `ADDN` and `DELN`. They are not used in the gradual growth method, and so, applying a similar procedure as in the case of method no. 2 was unfeasible.

Table 2. Results of additional experiments (1 - Average number of co-evolutionary cycles necessary to generate successful ANN)

	Method no. 2	Method no. 3	Method no. 4
1	47729.2	97611.9	85924.1

6. Summary

The paper compares methods whose main objective is to speed up AE when large ANNs are required to be built. The experiments were carried out in the predator-prey problem and they showed that for smaller ANNs, i.e. ANNs, which, as it turned out, were sufficient to solve the problem mentioned, the gradual growth method, i.e. the method used so far in AE, outperforms all the methods proposed in the paper. However, the experiments also showed that a part of the methods mentioned can be successfully combined with the classic variant of AE. First, they are used to find a rough size of ANN. Then, the gradual growth method is activated to complete the task of finding effective ANN. The experiments revealed that such an approach can be equally effective and, what is more important, can also be faster than the one used so far.

References

Cangelosi, A., Parisi, D. and Nolfi, S. (1994) Cell division and migration in a genotype for neural networks. *Network: Computation in Neural Systems* **5** (4), 497-515.

Gruau, F. (1994) *Neural Network Synthesis Using Cellular Encoding And The Genetic Algorithm.* PhD Thesis, Ecole Normale Superieure de Lyon.

Gruau, F. (1995) Automatic Definition of Modular Neural Networks. *Adaptive Behavior* **3**(2), 151-183.

Gruau, F., Whitley, D. and Pyeatt, L. (1996) A comparison between cellular encoding and direct encoding for genetic neural networks. In: J.R. Koza, D.E. Goldberg, D.B. Fogel and R.L. Riolo, eds., *Genetic Programming: Proceedings of the First Annual Conference.* MIT Press, 81-89.

Kitano, H. (1990) Designing neural networks using genetic algorithms with graph generation system. *Complex Systems* **4**, 461-476.

Krawiec, K. and Bhanu, B. (2005) Visual Learning by Coevolutionary Feature Synthesis. *IEEE Trans. on Systems, Man, and Cybernetics, Part B: Cybernetics* **35**, 409-425.

Luke, S. and Spector, L. (1996) Evolving Graphs and Networks with Edge Encoding: Preliminary Report. In: J.R. Koza, ed., *Late Breaking Papers at the Genetic Programming 1996 Conference.* Stanford University, CA,

USA. Stanford Bookstore (1996), 117-124.

Miller, G.F., Todd, P.M., and Hegde, S.U. (1989) Designing Neural Networks Using Genetic Algorithms. In: J.D. Schaffer, ed., *Proceedings of the Third International Conference on Genetic Algorithms*. Morgan Kaufmann, San Mateo, CA, 379-384.

Moriarty, D.E. and Miikkulainen, R. (1998) Forming Neural Networks Through Efficient and Adaptive Coevolution. *Evolutionary Computation* **5** (4), 373-399.

Moriarty, D.E. (1997) *Symbiotic Evolution of Neural Networks in Sequential Decision Tasks*. PhD thesis, The University of Texas at Austin, TR UT-AI97-257.

Nolfi, S. and Parisi, D. (1992) Growing neural networks. In: C.G. Langton, ed., *Artificial Life III*. Addison-Wesley, Reading, MA.

Nordin, P., Banzhaf, W. and Francone, F. (1999) Efficient Evolution of Machine Code for CISC Architectures using Blocks and Homologous Crossover. In: L. Spector, W. Langdon, U. O'Reilly and P. Angeline, eds., *Advances in Genetic Programming III*, MIT Press, 275-299.

Potter, M. (1997) *The Design and Analysis of a Computational Model of Cooperative Coevolution*. PhD thesis, George Mason University, Fairfax, Virginia.

Potter, M.A. and De Jong, K.A. (2000) Cooperative coevolution: An architecture for evolving coadapted subcomponents. *Evolutionary Computation* **8** (1), 1-29.

Praczyk, T. (2007a) Evolving co-adapted subcomponents in Assembler Encoding. *International Journal of Applied Mathematics and Computer Science* **17**(4).

Praczyk, T. (2007b) Procedure application in Assembler Encoding. *Archives of Control Science* **17(LIII)**,1, 71-91.

Praczyk, T. (2008) Modular networks in Assembler Encoding. *Computational Methods in Science and Technology, CMST* **14** (1), 27-38.

Appendix A – List of operations used in experiments

CHG – see Fig. 3.

CHGC0 – see Fig. 3.

CHGC1 – Update of a certain number of elements in a column. Index of the column, index of the first element in the column, the number of changed elements and the new value for the elements, the same for all of them, are located in parameters of the operation.

CHGC2 – Update of a certain number of elements in a column. The new value of each element is a sum of the parameter of the operation and the current value of the element. The second parameter of the operation is an index of

the column. The third and fourth parameters of the operation determine the number of changed elements and the index of the first element in the column to be changed, respectively.

CHGC3 – A number of elements from one column are transferred to another column. Both columns are indicated by the parameters of the operation. The number of the transferred elements and the index of the first element in the column transferred are also included in parameters of the operation.

CHGC4 – Update of a certain number of elements in a column. The new value of each element is a sum of the current value of this element and the respective value from memory of the program. The index of the column, the index of the first element in the column, the number of changed elements, and the pointer to data, where ingredients of individual sums are memorized, are located in parameters of the operation.

CHGR0 – like CHGC0, but the update refers to the row of the matrix.

CHGR1 – like CHGC1.

CHGR2 – like CHGC2.

CHGR3 – like CHGC3.

CHGR4 – like CHGC4.

CHGM0 – Update of a block of elements. Elements are updated in columns, in turn, one after another, starting from an element pointed by the parameters of the operation. The number of changed elements and the place in the memory where new values for elements are located are determined by parameters of the operation.

CHGM1 – like CHGM0, but the new value of every element is a sum of its current value and the parameter of the operation.

CHGM2 – like CHGM0, but the new value of each element is a sum of its current value and the value from the memory part of a program. The number of changed elements and the place in the memory where arguments of individual sums are located are determined by the parameters of the operation.

JMP – see Fig. 4.