Int. J. Appl. Math. Comput. Sci., 2007, Vol. 17, No. 4, 549–563
DOI: 10.2478/v10006-007-0045-9

amcs

EVOLVING CO-ADAPTED SUBCOMPONENTS IN ASSEMBLER ENCODING

TOMASZ PRACZYK

Institute of Radioelectronic Systems, Polish Naval Academy
ul. Śmidowicza 69, Gdynia, Poland
e-mail: T.Praczyk@amw.gdynia.pl

The paper presents a new Artificial Neural Network (ANN) encoding method called Assembler Encoding (AE). It assumes that the ANN is encoded in the form of a program (Assembler Encoding Program, AEP) of a linear organization and of a structure similar to the structure of a simple assembler program. The task of the AEP is to create a Connectivity Matrix (CM) which can be transformed into the ANN of any architecture. To create AEPs, and in consequence ANNs, genetic algorithms (GAs) are used. In addition to the outline of AE, the paper also presents a new AEP encoding method, i.e., the method used to represent the AEP in the form of a chromosome or a set of chromosomes. The proposed method assumes the evolution of individual components of AEPs, i.e., operations and data, in separate populations. To test the method, experiments in two areas were carried out, i.e., in optimization and in a predator-prey problem. In the first case, the task of AE was to create matrices which constituted a solution to the optimization problem. In the second case, AE was responsible for constructing neural controllers used to control artificial predators whose task was to capture a fast-moving prey.

Keywords: neural networks, evolution, neuroevolution.

1. Introduction

ANNs constitute a sub-domain of artificial intelligence that is broadly used to solve various problems in different fields (e.g., pattern classification, function approximation, optimization, image compression, associative memories, robot control problems, etc.). The performance of ANNs highly depends on two factors, i.e., the network topology and the set of network parameters (typically weights). Therefore, to develop an appropriate network, it is necessary to determine the architecture and parameters. There are many different ANN learning algorithms that change values of parameters leaving the structure completely intact. In such a case, the process of searching for a proper network topology is the task of a network designer who arbitrarily chooses the network structure, starts network learning and finally puts the network to a test. If the result of the test is satisfactory, the learning process is stopped. If not, it is continued. The designer manually determines the next potential network topology and runs the learning algorithm again. Such a loop-topology determination and learning procedure is repeated until the network, which is able to carry out a dedicated task at an appropriate level, is found. At a first glance, it is apparent that such a procedure could be very time-consuming and, what is worse, in the case of more complex problems it can lead to a situation when all chosen and trained networks would be incapable of solving the task.

In addition to the learning concept presented above, there exist other approaches that can be called constructive and destructive. The former use the learning philosophy that consists in an incremental development of the ANN starting from a small architecture. Initially, the ANN has a small number of components to which next components are gradually added until a resultant network fully meets the requirements imposed. In turn, the latter prepare a large fully connected ANN and then try to remove individual elements of the network, such as synaptic connections and neurons.

Genetic Algorithms (GAs) are the next technique that has been successfully applied in recent years to search for effective ANNs (Curran and O'Riordan, 2002; Floreano and Urzelai, 2000; Mandischer, 1993). A GA processes a population of genotypes that typically encode one phenotype, although encoding several phenotypes is also possible. In an ANN evolution, genotypes are encodings of the corresponding ANNs (phenotypes). The evolutionary procedure works by selecting genotypes (encoded networks) for reproduction based on their fitness, and then by introducing genetically changed offspring (mutation, crossover

and other genetic operators) into a newly created population. Repeating the whole procedure over many generations causes the population of encoded networks to gradually evolve into individuals that correspond to high fitness phenotypes (ANNs).

The paper presents a new ANN encoding method called Assembler Encoding (AE). AE originates from the cellular (Gruau, 1994) and edge encoding (Luke and Spector, 1996), although it also has features common with Linear Genetic Programming presented, among other things, in (Krawiec and Bhanu, 2005; Nordin *et al.,* 1999). In AE the network is represented as a structure similar to a simple assembler program. The Assembler Encoding Program (AEP) contains an executive part with operations, a part with data, and it operates on a Connectivity Matrix (CM) that indicates the strength of every interneuron connection. AE has many variants (Praczyk, 2007). Each variant uses a different AEP encoding method, i.e., the method used to represent the AEP in the form of a chromosome or a set of chromosomes, and a different method used to construct a modular ANN. The paper proposes a new AEP encoding scheme. It is an adaptation of the idea of evolving co-adapted subcomponents proposed by Potter and De Jong (Potter and De Jong, 1994; Potter and De Jong, 1995; Potter 1997; Potter and De Jong, 2000). The scheme proposed assumes a separate evolution of individual elements of AEPs, i.e., operations and data. Each AEP is composed of operations and data from various populations. The procedure of adding and replacing populations with operations and data is applied to regulate the length of AEPs.

The scheme proposed was tested on optimization and predator-prey problems. In the first case, the task of AEPs was not to construct ANNs but to build solutions to several optimization problems. In the second case, AEPs performed a task consistent with the main area of application of AE, i.e., they were used to create ANNs. ANNs were in turn responsible for the control of artificial predators whose task was to capture a fast moving prey.

The article is organized as follows: Related research is reviewed in the next section. Section 3 is a short introduction to AE. A detailed presentation of proposed concept is included in Sections 4–6. The results of the experiments for the optimization problem are presented in Section 7. Section 8 illustrates the results of experiments conducted for the predator-prey problem, and a summary is drawn in Section 9.

2. Related Work

In recent years, many attempts have been made to define genotypes for neural networks and to describe the genotype-into-phenotype mapping process. One of the earliest concepts was proposed by Miller *et al.* (1989). Their approach consists in the application of a Connectivity Matrix (CM). Each element of the matrix informs about the existence of a connection between two neurons or about the lack of such a connection.

Moriarty and Miikkulainen (1998) proposed a Symbiotic Adaptive NeuroEvolution (SANE). Their concept assumes that information necessary to create a network is included in two types of individuals, i.e., in blueprints and in neurons encoded. Both types of individuals evolve in separate populations. The task of blueprints is to record the most effective combinations of neurons. Each blueprint specifies a set of neurons that cooperate well together. The population of neurons includes individuals encoding hidden neurons of a two-layered feed-forward ANN. Each individual from the population of neurons defines connections of the neuron with input and output neurons and the strength of every connection.

Kitano (1990) defined the matrix rewriting encoding scheme. Initially, the method assumes 2×2 matrix that contains nonterminal elements. These elements are subsequently substituted for matrices including other nonterminal or terminal elements. This process is repeated until the resultant enlarged matrix contains only terminals that indicate the existence of a connection between neurons or the lack of such a connection.

In the Nolfi and Parisi model (Nolfi and Parisi, 1992), the genotype defines the location of each neuron in a two-dimensional space and growth parameters of neuron axons. The neurons that are in the left part of the space are considered to be input neurons and the ones placed in the right are considered to be output neurons. The remaining neurons are hidden neurons. After the location phase, axons of neurons start to grow further according to an assumed procedure. The connection between neurons is established if the branching axon of a source neuron reaches another neuron.

A natural continuation of Nolfi and Parisi's work is the concept proposed by Cangelosi *et al.* (1994). They decided to substitute the direct encoding of the location of neurons (in the chromosome) for the procedure of cell division and cell migration. One mother cell splits into "daughter" cells which, in turn, split into next cells. The division process is repeated for a number of generations after which all created cells become mature (become neurons). Apart from the division, the cells can be subjected to migration that consists in locating each cell near the mother cell. Once the division and migration procedure is completed, the axon growth phase occurs which runs in a similar way as in the scheme proposed by Nolfi and Parisi (1992).

The chromosome in Gruau's cellular encoding (Gruau, 1994; Gruau, 1995; Gruau *et al.,* 1996; Whitley *et al.,* 1995) contains a set of instructions that are applied to a network consisting initially of one hidden node. The network evolves towards larger structures during successive executions of individual instructions. The instruc-

tions are organized into a tree and include operations such as node duplication, node division, the removal of connectivity and many others. A very important feature of cellular encoding is its potential to build modular ANNs consisting of similar elements located in various places of a network. This potential is a result of applying a set of trees (with instructions) instead of applying a single tree, and repeated execution of instructions grouped in each of them. The result of such a procedure is analogous to multiple procedure execution in the main body of the structural program. Another crucial characteristic of cellular encoding is the form of the chromosome—a tree. Due to this feature the only evolutionary technique which is applicable to process individuals constructed in this way is genetic programming.

The related encoding method is edge encoding proposed by Luke and Spector (1996). Their scheme uses edge operators instead of node instructions. A network grows through adding, removing, and executing operations on edges, and not on nodes as was the case in cellular encoding. The remaining aspects of both encoding methods are conceptually very similar.

3. Fundamentals of Assembler Encoding

AE, like cellular and edge encodings, creates an ANN by means of a program. However, there are two significant differences between the above-mentioned schemes. Firstly, the chromosomes in AE are programs, procedures, operations or data encoded in the form of linearly ordered sequences of genes, while in the cellular and edge encoding chromosomes take the form of trees. Secondly, the execution of individual instructions in AE does not create a network directly, as in cellular and edge encodings. AEPs operate on the data structure which is Miller, Todd and Hedge's CM (Miller *et al.,* 1989). Initially, the CM is designed and once the AEP stops an appropriate network is constructed.

There are three key elements of AE: the AEP, the CM and two auxiliary registers. The AEP is an ordered set of procedures, which in turn are composed of a sequence of operations (code part of the procedure) and data (memory part of the procedure). The parameters of the procedures determine which part of the CM is altered by the procedure. Operations included in the procedures also possess parameters. The performance of the AEP consists in running all procedures in turn. Operations included in every procedure are executed one after another, changing elements of the CM (initially all elements in the matrix are set to 0, as there are not any connections between neurons). They alter one or more elements of the CM. The kind of change depends on the type of operation while the address of the change is located in the registers and parameters of the operation. A detailed analysis of the role of registers is presented in the section where the construction of modular ANNs is described. Once the execution of the AEP is finished, the ANN is created based on the CM generated by the program. Figure 2 depicts a diagram of AE.

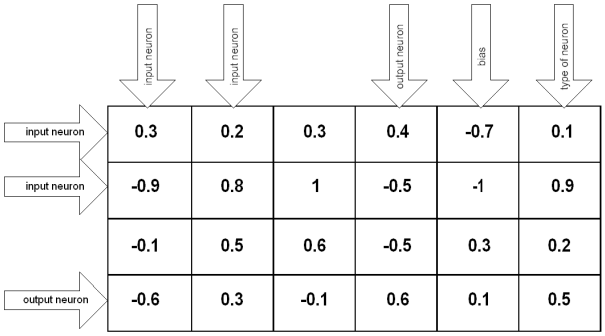

Fig. 1. Connectivity matrix.

The CM determines the ANN architecture. Each element of the matrix determines a synaptic weight between the corresponding neurons. For example, $component_{i,j}$ defines the link from neuron i to neuron j. Elements of the CM that are unimportant from the point of view of the process of ANN construction, e.g., because of the assumed feed-forward topology of the network, are neglected during ANN building. Apart from the basic part, the CM also possesses additional columns that describe neuron parameters, e.g., neuron type (sigmoidal, radial), bias, etc.

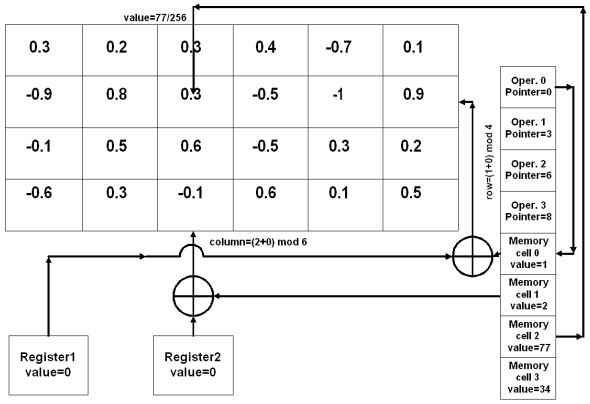

Fig. 2. Diagram of AE with a single procedure.

4. Operations

The basic task of operations is to change CM elements. The change can involve a single component or a larger set of matrix components. The simplest operation changes a single element in the matrix. The change location is determined in one of the parameters of the operation and

in registers while the change value is located in another parameter of the operation.

The exact implementation of the operation changing a single element of the CM is presented in Fig. 3. In the

```
CHG(p0, p1, p2, #)
{
row = (abs(p1) + R1) mod N;
column = (abs(p2) + R2) mod M;
C[row, column] = p0 / Max_value;
 }
```

Fig. 3. CHG operation changing a single CM element.

example, we assume that every operation can have maximally four parameters. Parameters that are unimportant for the implementation of the operation can be omitted and are marked with the *don't care* symbol "#". The following notation is used: $C[i, j]$ is an element of the CM, $i = 1, \ldots, N$, $j = 1, \ldots, M$, where N and M denote the size of the CM, R_i determine the value of the i-th register, $i = 1, 2$, *Max_value* is a scaling value, which scales all elements of the CM to the range $[-1, 1]$. Additionally, the following symbols will also be used: $D[i]$ – i-th datum in the memory part of the AEP, D_{Length} – number of memory cells.

As regards the operations that alter a larger group of CM elements, the following operations can be imagined: the change of the whole row or column, the change of a group of elements indicated by memory cells (and registers), the determination of elements of a given row (column) as the sum (difference) of other two rows (columns), the addition (subtraction) of some constant value to all elements of a row (column), etc. In the case of operations used to change a group of elements, information involving both the address of the change and the value of the change is usually placed in the memory. Each operation determines only a pointer indicating an address in the memory where this information is accessible. In order to illustrate the way the operations are constructed, two examples are presented.

Both examples present a column of the CM change operation. *CHGC6* fills the whole column indicated by p_0 and R_2 with a value from another column (pointed by p_1), whereas *CHGC0* uses data from memory. Here p_1 indicates the place in the memory part of the AEP where new values for the column elements are located.

To create an effective AEP consisting of operations presented above, it is necessary not only to find appropriate operations and data, but also to put them in a right sequence. Another approach is to exclusively use operations whose working effect does not depend on their sequence, e.g., operations whose outcome is a sum of the

```
CHGC0(p0, p1, p2, #)
{
column = (abs(p0) + R2) mod M;
numberOfIterations = abs(p2) mod N;
for(i = 0; i <= numberOfIterations; i++)
 {
 row = (i + R1) mod N;
 C[row, column] =
D[(abs(p1) + i) mod D.length] / Max_value;
  }
 }
```

Fig. 4. CHGC0 operation changing a part of the CM column.

```
CHGC6(p0, p1, #, #)
{
column1 = (abs(p0) + R2) mod M;
column2 = abs(p1) mod M;
for(i = 0; i < N; i++)
 {
 row = (i + R1) mod N;
 C[row, column1] = C[row, column2];
  }
 }
```

Fig. 5. CHGC6 operation changing the whole column of the CM.

value that constitutes a parameter of the operation and the value from the CM. (In this case the values of the CM are not scaled to an acceptable range until the whole program stops working). In this solution any sequence of operations in the AEP yields the same result (in fact, some additional assumptions have to be fulfilled to obtain such a result, see further). Examples of modifications of sequence-dependent operations are shown below.

```
CHG_1(p0, p1, p2, #)
{
row = (abs(p1) + R1) mod N;
column = (abs(p2) + R2) mod Z;
C[row, column] = C[row, column] + p0;
 }
```

Fig. 6. Modification of the CHG operation.

```
CHGC6_1(p0, p1, #, #)
{
column1 = (abs(p0) + R2) mod Z;
column2 = abs(p1) mod Z;
for(i = 0; i < N; i++)
 {
 row = (i + R1) mod N;
 C[row, column1] = C[row, column1] + C[row, column2];
  }
 }
```

Fig. 7. Modification of the CHGC6 operation.

5. Modular Networks

We propose two methods that make it possible to create modular networks. Both methods execute the same piece of code many times but in different places of the CM. The first method is a simple jump operation. It determines the place in the code part of the procedure where processing should continue (the jump operation is restricted to the part of the procedure that precedes the jump; only backward jumps are acceptable). It also determines the number of jumps and the place in the memory where new values of registers are placed. The construction of jump causes the same part of the code to be run in different locations of the CM, i.e., locations indicated by values of registers that are changed at the very start of the jump operation.

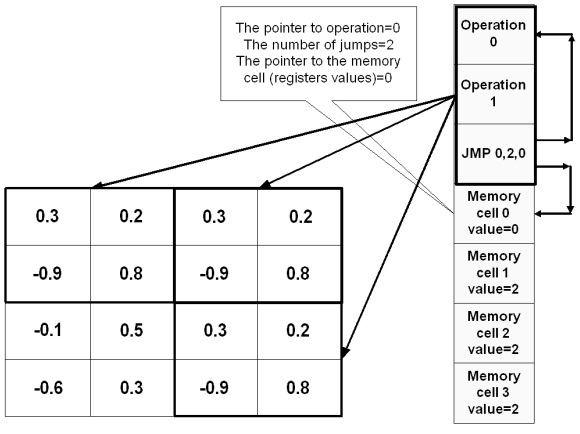

Fig. 8. Illustration of the jump operation.

Figure 8 shows the situation in which the jump operation denoted by JMP is run twice. The sequence of two operations (Operation 0 and Operation1) is executed three times, but each time in a different place of the CM. The first time, operations are executed for initial values of registers. The second time, after the first activation of the jump, registers are changed to $R_1 = 0$ and $R_2 = 2$. The last execution of the two operations is connected with the following values of registers: $R_1 = 2$, $R_2 = 2$.

The second method that makes it possible to create modular networks is the application of procedures. Each procedure can be run many times, each time in a different place of the CM. Repeated execution of the same procedure makes the effect of its work visible in many areas of the CM. Owing to the application of registers, the procedure can be executed in different regions of the CM. Every change in the CM is made with respect to them. In order to execute the procedure in different places of the CM, it suffices to change the values of registers beforehand. New values for registers are stored in the main program (AEP). The program executes procedures in sequence, changing the values of registers before invoking each of them.

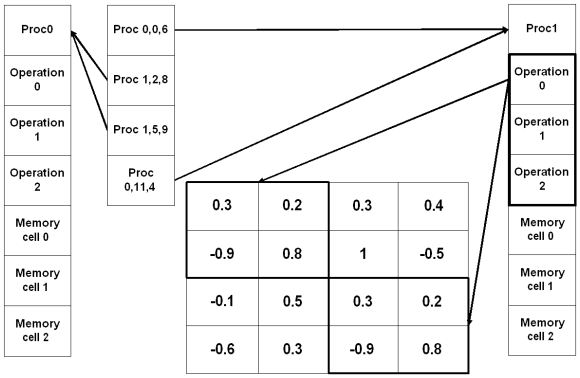

Fig. 9. Illustration of the procedure.

6. Encoding the AEP into the Chromosome(s)

In order to use evolutionary techniques to search for effective AEPs, it is required to present the whole information necessary to construct the program in the form of a chromosome or a set of chromosomes. The simplest AEP encoding scheme consists in placing the whole AEP in one chromosome (Fig. 10). Let us call this scheme as Scheme 1. In this solution, a single chromosome contains the whole information necessary to create the AEP, i.e., the initial size of the CM, the sequence of operations (a single-procedure AEP is assumed) and data. In order to know where the borderline between operations and data is, the chromosome includes an additional field storing this kind of information.

The next possibility of encoding the AEP is to locate its components in different chromosomes. For example, one population can store chromosome operations, the next one chromosome data and the last population can contain chromosome programs with pointers to individuals from

the remaining two populations (Fig. 11). Let us call the scheme described above Scheme 2.

This solution is similar to Moriarty and Miikkulainen's SANE approach (Moriarty and Miikkulainen, 1998), in which we have a population of blueprints and a population of neurons. Chromosome programs are equivalents of blueprints in the solution being considered that determine which operations and data cooperate well together, whereas chromosome operations and chromosome data are counterparts of neurons from the SANE which determine the partial architecture of the ANN.

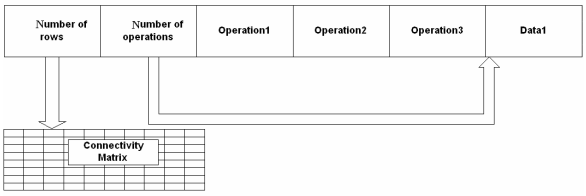

Fig. 10. AEP encoded into a single chromosome.

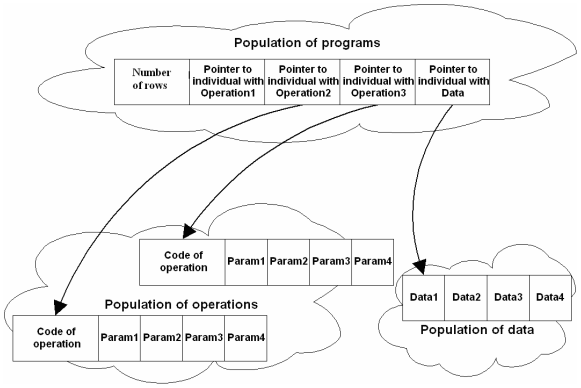

Fig. 11. AEP portioned into individuals from three different populations (programs, operations and data).

The next AEP encoding scheme, called Scheme 3, is a slight modification of Scheme 2. Whereas Scheme 2 uses sequence dependent operations, Scheme 3 is the only scheme presented in the paper which uses operations whose sequence does not affect the working effect of the AEP. To make AEPs completely independent of the sequence of operations, no change in the values of registers can take place in the middle of the run of the AEP. If such a change happened, different CMs could be produced by means of different sequences of operations. To prevent it, one copy of a jump is always located at the end of each AEP generated. Additionally, the jump mentioned always indicates the first operation in the AEP. This way, a single execution of the whole sequence of operations preceding the jump is always performed in the same area of the CM.

The AEP whose structure is depicted in Fig. 12 can be encoded in the way similar to that AEPs are produced with Scheme 2. In this case, to generate the AEP, the following set of chromosomes is required: chromosome-program, chromosome-operations, chromosome-jump-operation and chromosome-data. All chromosomes mentioned come from separate populations.

In this paper, we want to suggest another AEP encoding method whose main idea was borrowed from (Potter and De Jong, 1994; Potter and De Jong, 1995; Potter 1997; Potter and De Jong, 2000). To create the AEP the proposed scheme, called Scheme 4 (Fig. 13), combines operations and data from various populations. Each population of chromosome-operations has an assigned number determining the position of the operation from the population in the AEP. In this approach, the number of operations corresponds to the number of populations of chromosome-operations. Each population delegates exactly one representative to each AEP created. At the beginning, AEPs have only one operation and a sequence of data. Both the operation and data come from two different populations. Further populations of operations are successively added if the generated AEPs cannot accomplish an improvement in the performance over some assumed number of co-evolutionary cycles (we used the term "co-evolutionary cycle" to distinguish it from evolutionary generation that takes place inside a single population of operations and data).

Populations of operations and data can also be replaced by newly created populations. This may happen if the contribution of a population (contribution of operations from a population) to the creation of AEPs is considerably less than the contribution of the remaining populations. In our experiments, the contribution of a given population was measured as an average fitness of operations contained in that population.

The proposed approach makes it possible to generate many different AEPs as there are many combinations of operations from different populations. In order to restrict the number of possible AEPs generated in each co-evolutionary cycle, we used the solution proposed in (Potter, 1997). In each cycle, the best five individuals from each population are selected. These individuals are used in the next cycle to create AEPs. Each AEP is created based on the individual being currently evaluated and based on individuals belonging to a selected set of the best individuals from the previous cycle.

Five AEPs are generated for each individual evaluated. One program is produced based on the best individuals from the previous cycle. The remaining four programs are constructed based on random individuals from the set of the best individuals from the previous cycle. Because each individual participates in five different AEPs, each of them receives either the fitness of the best AEP in which it has taken part, or the average fitness of all of its five contributions.

7. Experiments on an Optimization Problem

AE is an ANN encoding scheme. It represents the ANN in the form of a linearly organized structure similar to an assembler program. Such a representation allows the application of GAs to search for better and better ANNs. In order to build an ANN, the AEP first creates a CM that is subsequently transformed into a resultant ANN. Such an action of AE in the process of the ANN construction and the application of the intermediate form, i.e., the CM, in this process makes it also possible to take advantage of AE in the optimization problem, in which the solution can be presented in the form of a matrix. Given this feature of AE, we decided first to test it just in the optimization problem. In this case, creating an ANN and checking its performance is not necessary. Therefore, from a technical point of view, tests are much easier to carry out than tests with the ANN participation. Even though searching for optimal matrices is not the target use of AE, experiments in this field can provide useful information about potentials of the encoding method proposed. Particularly interesting would be, for instance, knowledge concerning differences in the performance between various AEP encoding schemes. This knowledge can be employed in further experiments in which ANNs will be used to reduce the number of tested variants of AE only to those which have produced satisfactory results for the optimization problem. We can treat the application of AE to the optimization problem as the first stage and a starting point of further research in which the ability of the encoding scheme proposed to create effective ANNs will be verified.

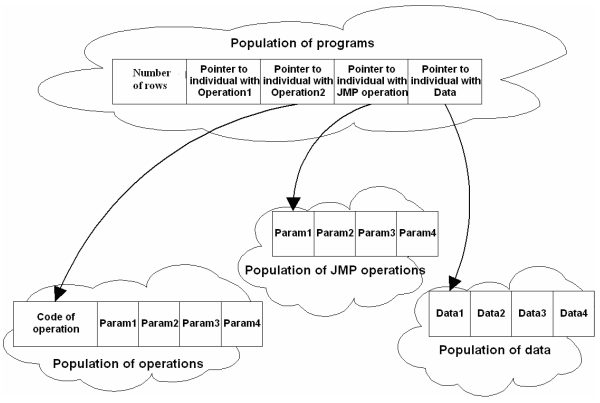

Fig. 12. AEP portioned into individuals from four different populations (population of programs, operations, jump operations and data).

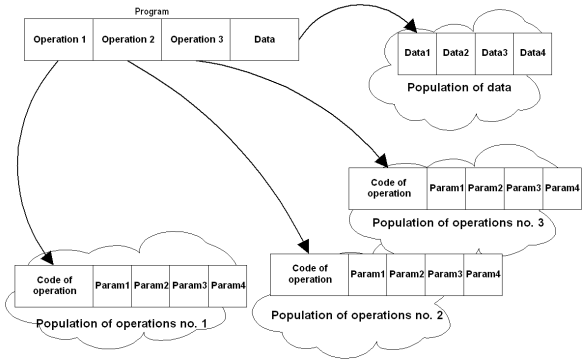

Fig. 13. Proposed concept of the AEP encoding scheme.

7.1. Tested Objective Functions. Five different test objective functions were used during the experiments:

$$f_1(\mathbf{C}) = -\left(1000 + \sum_{i=1}^{10}\sum_{j=1}^{10}\Big[\big(5.12(C[i,j]-0.2)\big)^2 - 10\cos\big(2\pi 5.12(C[i,j]-0.2)\big)\Big]\right),$$

$$f_2(\mathbf{C}) = -\left(1 + \sum_{i=1}^{10}\sum_{j=1}^{10}\frac{\big(5.12(C[i,j]-0.2)\big)^2}{4000} - \prod_{i=1}^{10}\prod_{j=1}^{10}\cos\left(\frac{512(C[i,j]-0.2)}{\sqrt{i+10(j-1)}}\right)\right),$$

$$f_k = -\sum_{i=1}^{10}\sum_{j=1}^{10}|A_k[i,j]|, \quad k = 3,4,5,$$

$$A_3[i,j] = \begin{cases} 1 - C[i,j], & i,j = 1\ldots 5,\\ 1 + C[i,j], & i,j = 6\ldots 10,\\ 0.4 - C[i,j], & i = 1\ldots 5,\ j = 6\ldots 10,\\ 0.4 + C[i,j], & i = 6\ldots 10,\ j = 1\ldots 5, \end{cases}$$

$$A_4[i,j] = \begin{cases} 0.4 - C[i,j] & \text{if } \big((i+j)\text{mod}2\big) = 0,\\ 0.4 + C[i,j] & \text{otherwise,} \end{cases}$$

$$A_5[i,j] = \begin{cases} 0.2 - C[i,j] & \text{if } i = j,\\ 0.2 + C[i,j] & \text{if } i = 10 - j + 1,\\ 0.4 - C[i,j] & \text{if } i = 6,\ i \neq j\\ & \text{and } i \neq 10 - j + 1,\\ 0.4 + C[i,j] & \text{if } i = 6,\ i \neq j\\ & \text{and } i \neq 10 - j + 1,\\ C[i,j] & \text{otherwise.} \end{cases}$$

$$\mathbf{C}^{\mathbf{3}}_{\mathbf{opt}} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0.4 & 0.4 & 0.4 & 0.4 & 0.4 \\ 1 & 1 & 1 & 1 & 1 & 0.4 & 0.4 & 0.4 & 0.4 & 0.4 \\ 1 & 1 & 1 & 1 & 1 & 0.4 & 0.4 & 0.4 & 0.4 & 0.4 \\ 1 & 1 & 1 & 1 & 1 & 0.4 & 0.4 & 0.4 & 0.4 & 0.4 \\ 1 & 1 & 1 & 1 & 1 & 0.4 & 0.4 & 0.4 & 0.4 & 0.4 \\ 0.4 & 0.4 & 0.4 & 0.4 & 0.4 & -1 & -1 & -1 & -1 & -1 \\ 0.4 & 0.4 & 0.4 & 0.4 & 0.4 & -1 & -1 & -1 & -1 & -1 \\ 0.4 & 0.4 & 0.4 & 0.4 & 0.4 & -1 & -1 & -1 & -1 & -1 \\ 0.4 & 0.4 & 0.4 & 0.4 & 0.4 & -1 & -1 & -1 & -1 & -1 \\ 0.4 & 0.4 & 0.4 & 0.4 & 0.4 & -1 & -1 & -1 & -1 & -1 \end{bmatrix}, \tag{1}$$

$$\mathbf{C}^{\mathbf{4}}_{\mathbf{opt}} = \begin{bmatrix} 0.4 & -0.4 & 0.4 & -0.4 & 0.4 & -0.4 & 0.4 & -0.4 & 0.4 & -0.4 \\ -0.4 & 0.4 & -0.4 & 0.4 & -0.4 & 0.4 & -0.4 & 0.4 & -0.4 & 0.4 \\ 0.4 & -0.4 & 0.4 & -0.4 & 0.4 & -0.4 & 0.4 & -0.4 & 0.4 & -0.4 \\ -0.4 & 0.4 & -0.4 & 0.4 & -0.4 & 0.4 & -0.4 & 0.4 & -0.4 & 0.4 \\ 0.4 & -0.4 & 0.4 & -0.4 & 0.4 & -0.4 & 0.4 & -0.4 & 0.4 & -0.4 \\ -0.4 & 0.4 & -0.4 & 0.4 & -0.4 & 0.4 & -0.4 & 0.4 & -0.4 & 0.4 \\ 0.4 & -0.4 & 0.4 & -0.4 & 0.4 & -0.4 & 0.4 & -0.4 & 0.4 & -0.4 \\ -0.4 & 0.4 & -0.4 & 0.4 & -0.4 & 0.4 & -0.4 & 0.4 & -0.4 & 0.4 \\ 0.4 & -0.4 & 0.4 & -0.4 & 0.4 & -0.4 & 0.4 & -0.4 & 0.4 & -0.4 \\ -0.4 & 0.4 & -0.4 & 0.4 & -0.4 & 0.4 & -0.4 & 0.4 & -0.4 & 0.4 \end{bmatrix}. \tag{2}$$

A 10×10 matrix constitutes the solution to these test functions. In all the cases presented above, the task of AEPs was to find a matrix that would maximize the function being currently optimized. The global maximum for all test functions is zero. The functions f_1 and f_2 are modifications of common test functions appearing in the literature dedicated to genetic algorithms. Original versions of these functions have maximum at $\mathbf{x} = (0, 0, \dots, 0)$. In the case of AE, the matrix consisting only of zeros is very simple to generate. Such a matrix may, e.g., result from an AEP which does nothing. Therefore, the original functions were altered so that the matrix including all the elements equal to 0.2 could constitute an optimal solution. The remaining functions, i.e., f_3, f_4 and f_5, were constructed so as to test how different variants of AE are able to make use of their potential to create modular ANNs. To create optimal matrices for these functions, AEPs can use two methods. The first method uses the so-called "brute force", i.e., it creates matrices by means of a large number of operations. Another method intelligently uses procedures or jumps. Optimal matrices for the functions f_3,f_4 and f_5 are given by (1)–(3).

7.2. Experimental Setup. During research, a canonical genetic algorithm was used to process populations of operations and data. In the experiments, we assumed a constant length of chromosome operations. Each chromosome operation included five blocks of genes. The first block determined a code of the operation (e.g., binary 00000 indicated that we deal with the CHG operation) while the remaining blocks contained a binary representation of four parameters of the operation. The list of operations applied is presented at the end of the paper. Chromosome data could change their length during consecutive co-evolutionary cycles. In order to make such a change possible, in addition to crossover and mutation, the genetic algorithm that processed the population of data used a cut-splice operator. The implementation of crossover applied in the experiments always produced offspring of the same length as parents. The cut-splice operator always activated after crossover and mutation modified the size of the chromosome through the addition or removal of a single block of genes (single data) from the same end of the chromosome.

In the experiments we used chromosome operations and chromosome data that consisted of 5-bit blocks of genes. Therefore, every chromosome operation used in the experiments included a total of 25 genes (5 blocks$\times$ 5 bits per block). In turn, chromosome data consisted of at least five genes (a single datum) and of at most 50 genes (10 data). Each use of an excessive number of data caused a drastic decrease in the AEP fitness. In the experiments, we assumed the maximal number of operations which could be included in the AEP, i.e., 12 operations. Initially, each

$$\mathbf{C}^{\mathbf{4}}_{\mathbf{opt}} = \begin{bmatrix} 0.2 & 0 & 0 & 0 & 0 & -0.4 & 0 & 0 & 0 & -0.2 \\ 0 & 0.2 & 0 & 0 & 0 & -0.4 & 0 & 0 & -0.2 & 0 \\ 0 & 0 & 0.2 & 0 & 0 & -0.4 & 0 & -0.2 & 0 & 0 \\ 0 & 0 & 0 & 0.2 & 0 & -0.4 & -0.2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.2 & -0.2 & 0 & 0 & 0 & 0 \\ 0.4 & 0.4 & 0.4 & 0.4 & -0.2 & 0.2 & 0.4 & 0.4 & 0.4 & 0.4 \\ 0 & 0 & 0 & -0.2 & 0 & -0.4 & 0.2 & 0 & 0 & 0 \\ 0 & 0 & -0.2 & 0 & 0 & -0.4 & 0 & 0.2 & 0 & 0 \\ 0 & -0.2 & 0 & 0 & 0 & -0.4 & 0 & 0 & 0.2 & 0 \\ -0.2 & 0 & 0 & 0 & 0 & -0.4 & 0 & 0 & 0 & 0.2 \end{bmatrix}. \quad (3)$$

AEP contained one operation and a set of data from two different populations. Consecutive populations of operations were added every 2000 or 5000 of co-evolutionary cycles if generated programs were not able to make an improvement in the performance within this period. Populations of operations and data could also be replaced by newly created populations when the contribution of substituted population to created programs was considerably less than the contribution of the remaining populations. In our experiments, the contribution of a population was measured as the average fitness of operations belonging to that population.

The remaining values essential for the conducted experiments are presented below:

- population size: 20 individuals,
- crossover probability: 0.7,
- mutation probability: 0.1, 0.01,
- cut-splice probability: 0.1 (in the case of chromosome data),
- number of co-evolutionary cycles: 50 000,
- number of AEPs generated for each test function: 30.

7.3. Experimental Results. As it turned out, the first two test problems, i.e., f_1 and f_2, were very easy to solve. Optimal matrices for these functions were found very quickly and, what is more, they were generated by means of very simple AEPs. Even though the functions f_1 and f_2 were constructed to have many local optima, this did not prevent the AEPs from finding optimal solutions. It seems that the main reason for such a situation is the homogeneity of the optimal matrices in the problems f_1 and f_2. All elements of these matrices equal 0.2. To create such matrices, only one operation, e.g., *CHGM0* (see Appendix 1), and only one datum containing the value of 0.2 are necessary.

The remaining problems, i.e., f_3, f_4 and f_5, were considerably more difficult to solve than the problems f_1 and f_2. This time, to find optimal matrices, more complex AEPs were necessary. What is more, in the case of f_5 no AEP encoding scheme could produce an AEP able to create an optimal matrix. Detailed results of the application of all AEP encoding schemes presented in Section 6 to optimize all test functions are presented in Table 1. Each cell in the table includes the following: the average result obtained for 30 evolutionary runs (top value), the result of the best program (middle value) and the sum of the number of operations and data in the best program (number of operations + number of data; bottom value). For the sake of comparison, in addition to the outcomes of the AEP proposed encoding scheme (Scheme 4) Table 1 also includes the performance of schemes tested previously (Praczyk, 2007) and presented in Section 6.

Table 1. Results of the optimization of all test functions.

	Scheme 1	Scheme 2	Scheme 3	Scheme 4
f_1	0	0	0	0
	0	0	0	0
	$2+2$	$1+2$	$1+2$	$1+2$
f_2	0	0	0	0
	0	0	0	0
	$1+2$	$3+2$	$3+2$	$1+2$
f_3	-18	-7.9	-4.9	-0.2
	-11.1	0	-0.5	0
	$9+5$	$11+5$	$12+10$	$5+8$
f_4	-14.5	-0.4	-2.3	0
	-4.8	0	0	0
	$9+4$	$5+2$	$12+2$	$7+3$
f_5	-3.6	-2.1	-2.4	-0.66
	-3.2	-1.5	-1.8	-0.6
	$11+4$	$9+3$	$12+1$	$12+6$

The experiments showed that Scheme 4 is undoubtedly the best AEP encoding scheme out of all the schemes tested so far. In all test problems, AEPs generated based on Scheme 4 outperformed programs generated by

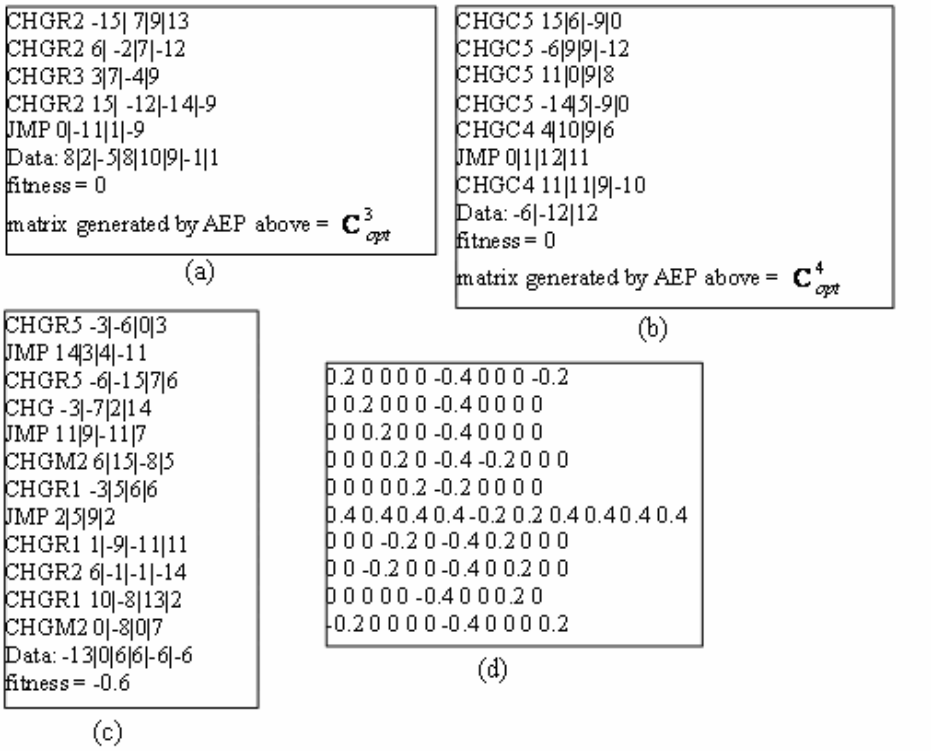

Fig. 14. (a) best AEP for f_3, (b) best AEP for f_4, (c) best AEP for f_5, (d) matrix generated by the AEP of (c).

means of the remaining schemes. The only problem in which AEPs encountered problems in generating the optimal matrix was the problem f_5. In the remaining cases, Scheme 4 was able to create a number of programs producing optimal matrices. The superiority of Scheme 4 over the remaining schemes is particularly apparent in the problem f_5 which, as it turned out, was the most difficult problem out of all the problems tested. The matrices created by AEPs were in this case considerably closer to the optimal matrix than the matrices generated by the remaining programs. Examples of AEPs generated during the experiments by Scheme 4 are presented in Fig. 14.

8. Experiments with ANNs

The next problem in which the proposed AEP encoding scheme was tested is a simple version of the predator-prey problem. This time, the task of AEPs was to generate ANNs controlling a set of cooperating predators whose common goal was to capture a fast moving prey. The main goal of the research reported in this section was only to check whether AE with the application of Scheme 4 is able to create simple ANNs. To find out true capabilities of AE and Scheme 4 with respect to creating ANNs, further experiments are required.

8.1. Environment. The predators and the prey used in the experiments lived in a common environment. We used a 20×20 square without obstacles but with two barriers located on the left and right sides of the square to represent the environment. Both of the barriers caused the predators as well as the prey to move right or left only to the point at which they reached one of the barriers. Moving further in the barrier direction did not cause any effect. In order to ensure an infinite space for the predators and the prey and for their struggles, we made the environment open at the bottom and at the top. This means that every attempt of a movement beyond the upper or lower borders of the square caused the object making such an attempt to move to the opposite side of the environment. As a result, the simple strategy of predators consisting in chasing the prey did not work. In such a situation, in order to evade the predators the prey could simply escape up or down.

8.2. Residents of the Artificial World. In our experiments, two predators and one prey coexisted in the artificial environment. The predators were controlled by the ANN produced by the AEP. They could select five actions: to move in the North, South, West, East directions or to stand still. The length of the step of every predator was 1 unlike the step of the prey, which was 2. In order to capture the prey, the predators had to cooperate. Their speed was half the speed of the escaping prey so they could not

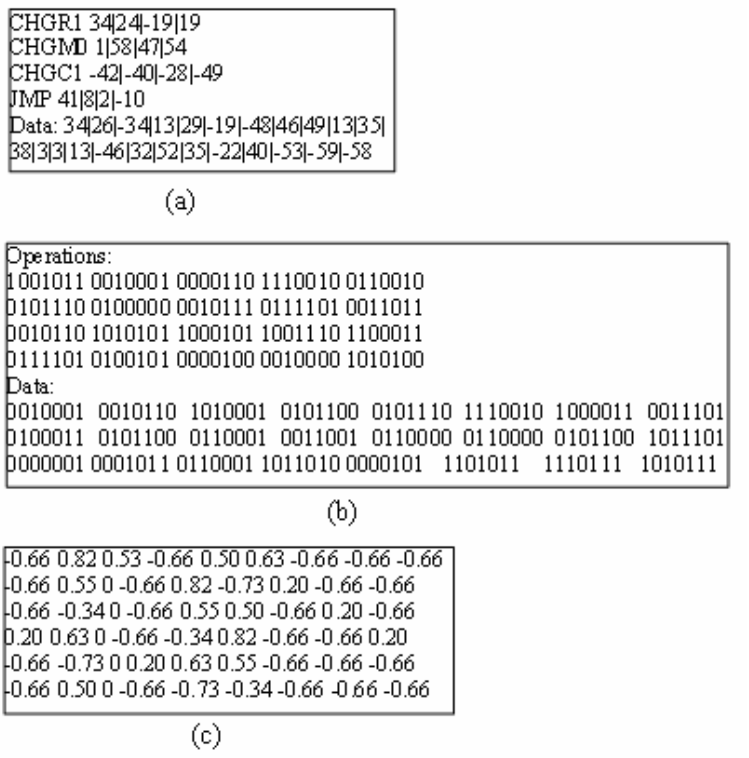

Fig. 15. (a) AEP, (b) its encoded form, (c) CM generated by the AEP of (a) and (b).

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 assumed that the predators could see the whole environment. The predators based the decision which actions to select in the prey's relative location with reference to each of them. In order to perform the task, the ANN controlling the predators had to possess four inputs and two outputs. Network outputs provided decisions to the predators whereas the inputs informed them about the prey's location with respect to each of them.

The simple prey was controlled by a simple algorithm which forced it to move directly away from the nearest predator but solely in the situation when the distance between it and the nearest predator was less than or equal to 5. In the remaining cases, i.e., when neither predator was closer to the prey than the assumed distance, the prey did not move. In the situation when the prey's action would cause hitting the barrier, another move was chosen. An alternative move prevented the prey from hitting the wall, and at the same time it maximally increased the distance between the prey and the nearest predator. When the prey was running away, it could select four actions: move in the North, South, West or East directions.

8.3. Neural Controllers. Connection matrices generated during the experiments by AEPs usually represented recurrent neural networks. However, we decided that the controllers used during the experiments have feed-forward architectures. In order to obtain such networks, we used only elements of CMs localized in their upper parts. The remaining elements were neglected during the process of neural network construction.

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

8.4. Experimental Setup. The experiments with ANNs took place in almost identical conditions as the experiments in the optimization problem (see Section 9.2). The only differences involved the construction of AEPs. In order to adjust AEPs to a new, more challenging task, two modifications were introduced to their construction. First, all of them were encoded in the form of chromosomes built of 7-bit blocks of genes. Previously, 5-bit blocks were in use. Second, all programs obtained a permission to have 20 data instead of 10 data as was the case previo-

usly. As regards the remaining parameters of the experiment, they remained the same as in the experiments for the optimization problem.

8.5. Evaluation Process. To evaluate generated networks, eight different scenarios were built. They differed in the initial location of the predators and the prey in the environment:

- Scenario 1: prey(10,10), predator1(0,0), predator2(20,20),
- Scenario 2: prey(10,10), predator1(20,20), predator2(0,0),
- Scenario 3: prey(10,10), predator1(0,10), predator2(20,10),
- Scenario 4: prey(10,10), predator1(10,0), predator2(10,20),
- Scenario 5: prey(10,10), predator1(0,0), predator2(20,0),
- Scenario 6: prey(0,20), predator1(10,0), predator2(20,10),
- Scenario 7: prey(0,10), predator1(10,1), predator2(10,19),
- Scenario 8: prey(0,20), predator1(0,10), predator2(10,20).

The tests proceeded in the following way: At first, each network was tested using Scenario 1. If the predators controlled by a network could not capture the prey during some assumed period, the test was stopped and the network received appropriate evaluation that depended on the distance between the prey and the nearest predator. However, if the predators grasped the prey, they were put to test in accordance with the next scenario. During the experiments, we assumed that the predators could perform 100 steps before the scenario was interrupted. To evaluate the networks, we used the following fitness function:

$$f(Network) = \sum_{i=1}^{n} f_i,$$

$$f_i = \begin{cases} d_{\max} - \min(d_1, d_2) & \text{if the prey not} \\ & \text{captured in Scenario } i, \\ f_{\text{captured}} + \frac{1}{a}(100 - s_i) & \text{if the prey} \\ & \text{captured in Scenario } i, \\ 0 & \text{if the prey not captured} \\ & \text{in the previous scenario,} \end{cases}$$

where the following notation is used:
f_i – reward received in Scenario i,

$d_{\max}$ – maximal distance between two points in the environment applied,

d_1, d_2 – distance between the prey and the first and second predators,

f_{captured} – reward for grasping the prey in a single scenario (in our experiments f_{captured} amounted to 100),

s_i – number of steps which the predators needed to capture the prey ($s_i < 100$),

a – this value prevents the situation in which a partial success would be better than a success in all scenarios,

n – number of scenarios (in our case $n = 8$).

8.6. Experimental Results. In order to test the AEP encoding scheme proposed in the predator-prey domain, 20 runs of the evolutionary process were performed. As a result of the conducted experiments, it turned out that all runs were successful, i.e., they produced AEPs that generated ANNs which, in turn, resulted in capturing the prey. Detailed results of the experiments conducted are presented in Table 2.

An exemplary program produced in this phase of experiments, its encoded form, the CM representing the network and the behavior of the predators controlled by the network are shown in Fig. 15.

The matrix depicted in Fig. 15 corresponds to the network consisting of six neurons (four input neurons and two output neurons). Three extra columns determine the types of individual neurons, their bias and values of the parameters of the neurons, e.g., the shape of the radial transfer function.

9. Summary

The article presents a new ANN encoding scheme called Assembler Encoding. The proposed encoding scheme represents the ANN in a very compact form, which allows applying genetic algorithms to create effective ANNs. Like cellular encoding and edge encoding, AE encodes an ANN in the form of a program. This permits building very complex and large neural architectures by means of relatively small chromosomes. Unlike cellular and edge encoding programs, which are represented as a tree, the AEP is a linearly ordered set of operations and data. Another difference between cellular, edge and assembler encodings is the object which is altered by the program. Cellular and edge encodings operate directly on a prototype of the ANN. AE creates a network indirectly, changing the CM that represents the ANN.

In addition to the short presentation of AE, the paper also suggests a new AEP encoding method which is an adaptation of the idea of evolving co-adapted subcomponents proposed by Potter and De Jong (Potter and De

Table 2. Results of application of Scheme 4 to the predator-prey problem.

average fitness (best fitness)	average number of neurons in the successful ANN (minimal number of neurons)	average length of the successful AEP, number of orders + number of data (shortest AEP)	average number of co-evolutionary cycles necessary to generate a successful AEP (minimal number of co-evolutionary cycles)
831.09 (865.34)	6 (6)	4.6+15.1 (4+5)	25670.3 (17277)

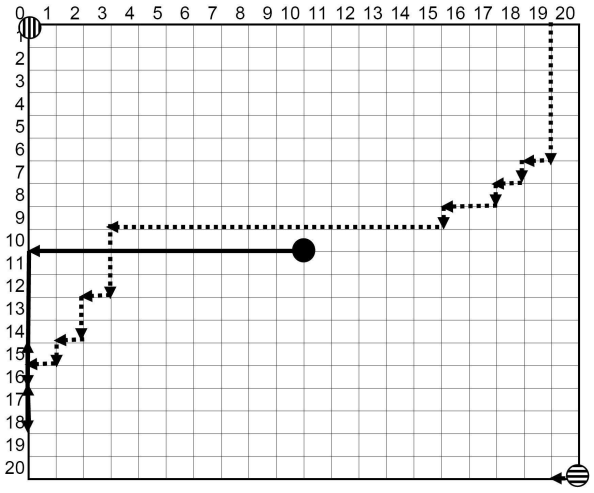

Fig. 16. Examplary behavior of the predators and the prey in Scenario 1. Circles determine initial positions of the predators and the prey (a black circle – the prey, a circle with vertical stripes – Predator 1, a circle with horizontal stripes – Predator 2) while arrows indicate directions of their movements (solid line – the prey, dashed line – Predator 1, dotted line – Predator 2).

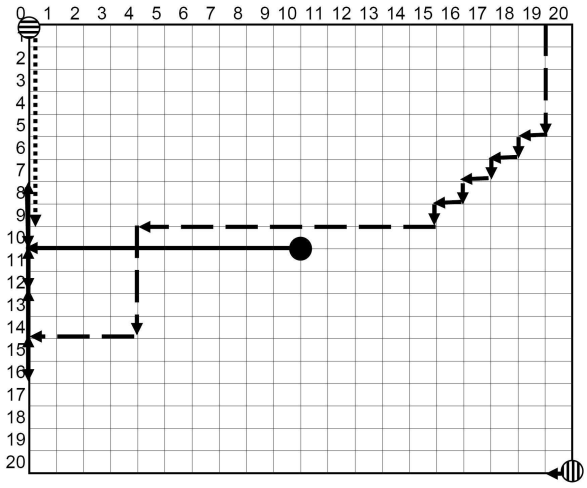

Fig. 17. Exemplary behavior of the predators and the prey in Scenario 2.

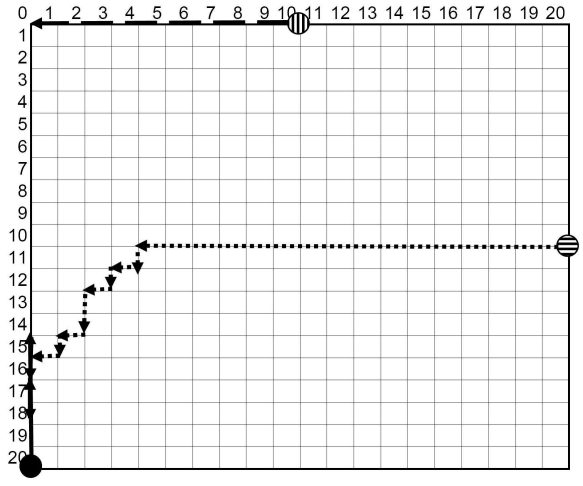

Fig. 18. Exemplary behavior of the predators and the prey in Scenario 6.

Jong, 1994; Potter and De Jong, 1995; Potter, 1997; Potter and De Jong, 2000). The proposed method assumes that each element of the AEP, i.e., each operation and a sequence of data, evolves in a separate population. To create the AEP, representatives of each population are selected and combined together. Each population delegates exactly one representative. Operations in the AEP are ordered according to numbers assigned to their populations. Sequences of data formed as a result of the evolution are always located at the end of the AEP.

The AEP encoding method proposed in the paper was tested on an optimization problem and a simple version of the predator-prey problem. In the case of the optimization problem, the proposed scheme turned out to be more effective than the remaining schemes described in the paper and tested within the framework of prior investigations (Praczyk, 2007). In the experiments with ANNs, the proposed method showed that it is able to produce effective ANNs. All ANNs generated during the tests were

successful. In all the tested cases, the predators controlled by ANNs captured the prey. What is more, all ANNs were created by relatively short AEPs. To represent an ANN consisting of six neurons, the shortest AEP required half of the genes required by the CM for the same purpose.

Generally, it is necessary to state that the results of the experiments presented in the paper are very encouraging. The tests showed that AE is able to solve simple optimization problems and, what is even more important, it is also able to construct simple and efficient ANNs. To find out true capabilities of AE in solving more complex problems, further experiments are required. In addition to the use of AE for more challenging tasks, future research will also include issues such as the search for new types of operations for AEPs, testing multi-procedure AEPs, the application of AE to construct ANNs with the Hebb self-organization, etc.

References

Cangelosi A., Parisi D. and Nolfi S. (1994): *Cell division and migration in a genotype for neural networks*. Network: Computation in Neural Systems, Vol. 5, No. 4, pp. 497–515.

Curran D. and O'Riordan C. (2002): *Applying evolutionary computation to designing networks: A study of the state of the art.* Technical Report NUIG-IT-111002, National University of Ireland.

Floreano D. and Urzelai J. (2000): *Evolutionary robots with online self-organization and behavioral fitness.* Neural Networks Vol. 13, No. 13, pp. 431–443.

Gruau F. (1994): *Neural network synthesis using cellular encoding and the genetic algorithm.* Ph.D. thesis, Ecole Normale Supērieure de Lyon.

Gruau F. (1995): *Automatic definition of modular neural networks.* Adaptive Behavior Vol. 3, No. 2., pp. 151–183.

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

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

Krawiec, K. and Bhanu, B. (2005): *Visual learning by coevolutionary feature synthesis.* IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics. Vol. 35, pp. 409–425.

Luke S. and Spector L. (1996): *Evolving graphs and networks with edge encoding: Preliminary report.* In: Late Breaking Papers at the Genetic Programming 1996 Conference, (J. R. Koza, Ed.), Stanford University, CA: Stanford Bookstore, pp. 117–124.

Mandischer M. (1993): *Representation and evolution of neural networks.* In: Artificial Neural Nets and Genetic Algorithms, (Albrecht R. F., Reeves, C. R., Steele U. C., Eds.), 643–649, Springer Verlag, New York.

Miller G.F., Todd P.M. and Hegde S.U. (1989): *Designing neural networks using genetic algorithms.* Proceedings of the 3rd International Conference on *Genetic Algorithms.* San Mateo, CA, USA, 379–384.

Moriarty D. E. and Miikkulainen R., (1998): *Forming neural networks through efficient and adaptive coevolution.* Evolutionary Computation, Vol. 5, No. 4, pp. 373–399.

Moriarty D. E. (1997): *Symbiotic evolution of neural networks in sequential ddecision tasks.* Ph.D. thesis, The University of Texas at Austin, TR UT-AI97-257.

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

Nordin P., Banzhaf W. and Francone F. (1999): *Efficient evolution of machine code for CISC architectures using blocks and homologous crossover.* In: Advances in Genetic Programming III (L. Spector and W. Langdon and U. O'Reilly and P. Angeline Eds.), MIT Press, Cambridge, MA, USA, pp. 275–299.

Potter M. (1997):*The design andanalysis of a computational model of cooperative coevolution.* Ph.D. thesis, George Mason University, Fairfax, VA.

Potter M. and De Jong K. A. (1995): *Evolving neural networks with collaborative species.* In: Proceedings of the 1995 Summer Computer Simulation Conference, (T. I. Oren, L. G. Birta, Eds.), Ottawa, Canada, The Society of Computer Simulation, pp. 340–345.

Potter M. A. and De Jong K. A. (1994): *A cooperative coevolutionary approach to function optimization.* In: The Third Parallel Problem Solving from Nature, Berlin: Springer-Verlag, pp. 249–257.

Potter M. A. and De Jong K. A. (2000): *Cooperative coevolution: An architecture for evolving coadapted subcomponents.* Evolutionary Computation. Vol. 8, No. 1, pp. 1–29.

Praczyk T. (2007) *Application of assembler encoding to optimization problem.* (submitted)

Whitley D., Gruau F. and Pyeatt L. (1995): *Cellular encoding applied to neurocontrol.* Proceedings of the 6-th International Conference on *Genetic Algorithms,* San Francisco, CA, USA, pp. 460–467.

Appendix. List of operations used in experiments

CHG: Update of an element. Both the new value and the element address are located in the parameters of the operation.

CHGC0: Update of some elements in a column. The index of the column, the index of the first element in the column that will be changed, the number of changed elements and a pointer to data, where new values of elements are memorized, are located in the parameters of the operation.

CHGC1: Update of some elements in a column. The index of the column, the index of the first element in

the column that will be changed, the number of changed elements and the new value for the column elements, the same for all elements, are located in the parameters of the operation.

CHGC2: Update of some elements in a column. The new value of every element is the sum of the operation parameter and the current value of this element. The second parameter of the operation is the index of the column. The third and fourth parameters of the operation determine respectively the number of changed elements and the index of the first element in the column that will be changed.

CHGC3: Part of elements from one column are transformed to another column. Both columns are indicated by the parameters of the operation. The number of transferred elements and the index of the first element in the column that will be transferred are also included in the parameters of the operation.

CHGC4: Update of some elements in a column. The new value of every element is the sum of the actual value of this element and the respective value from program memory. The column index, the index of the first element in the column that will be changed, the number of changed elements and a pointer to data, where ingredients of individual sums are memorized, are located in the parameters of the operation.

CHGR0: like **CHGC0**, but the corresponding update refers to a row of a matrix.

CHGR1: like **CHGC1.**

CHGR2: like **CHGC2**.

CHGR3: like **CHGC3**.

CHGR4: like **CHGC4**.

CHGM0: Change of a block of elements. The elements are updated in columns, in turn, one after another, starting from the 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 the parameters of the operation.

CHGM1: Like **CHGM0**, but a new value of every element is the sum of its current value and the parameter of operation.

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

JMP: Jump operation. The number of jumps, a pointer to the next operation and new values of registers are located in the parameters of the jump operation.

Received: 11 October 2006
Revised: 26 April 2007
Re-revised: 3 June 2007