Charles University in Prague
Faculty of Mathematics and Physics

DIPLOMA THESIS

Pavel Kasík

Evolutionary algorithms for structural learning of neural networks

Department of Theoretical Computer Science and Mathematical Logic

Diploma thesis supervisor: Mgr. Roman Neruda, CSc.,
Institute of Computer Science of the ASCR, v. v. i.

Study programme: Informatics

2010

I would like to express my thanks to diploma thesis supervisor Roman Neruda for guiding me and giving me valuable advices during my work.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

I declare that I wrote my diploma thesis independently and exclusively with the use of cited sources. I agree with lending the thesis.

Prague, 1st August 2010 Pavel Kasík

Contents

1 Artificial Neural Networks and Evolutionary Algorithm 8
1.1 Artificial Neural Network 8
1.2 Evolutionary Algorithm 10

2 Description and Analysis of NEAT 13
2.1 NEAT 13
2.2 Genotype 14
2.3 Mutations and Innovation number 14
2.4 Crossover 15
2.5 Speciating and Fitness 16
2.6 Incremental Growth 17

3 New Algorithm 19
3.1 Motivation 19
3.2 Node Naming 20
3.3 Crossover 22
3.4 Minor Modifications 24

4 Agent Parallel Implementation 26
4.1 JADE 26
4.2 Agent's structure design 27
4.3 Synchronization 28
4.4 Algorithm Parallelization 29

5 Description of New Algorithm Implementation 30
5.1 Genotype 30
5.2 Mutations 31
5.3 Selection and Crossover 32
5.4 Computation Agent 33

5.5 Synchronization Agent 35

6 Experiments **36**
6.1 XOR Experiment 36
6.2 Building Experiment 38
6.3 Cancer Experiment 40

7 Conclusion **42**
7.1 Recapitulation 42
7.2 Future work 43

References **44**

Název práce: Evoluční algoritmy pro strukturální učení neuronových sítí
Autor: Pavel Kasík
Katedra (ústav): Katedra teoretické informatiky a matematické logiky
Vedoucí bakalářské práce: Mgr. Roman Neruda, CSc.,
Ústav informatiky AV ČR, v.v.i.
e-mail vedoucího: Roman.Neruda@mff.cuni.cz

Abstrakt: Návrh topologie neouronových sítí je velmi komplikovaný problém, zejména když opustíme oblast standartních vrstevnatých sítí. Zajímavé řešení tohoto problému nám může poskytnout evoluční algoritmus. Jeden z možných evolučních algoritmů pro evoluci neuronových sítí je algoritmus NEAT. Cílem této práce je modifikovat a vylepšit schopnosti algoritmu NEAT. Vylepšení jsou zaměřena na využití polohy neuronu ve struktuře sítě, zlepšení křížení a představení možnosti paralelizace algoritmu zachovávající jeho ideje i ideje NEATu.

Klíčová slova: NEAT, evoluce, neuronová síť

Title: Evolutionary algorithms for structural learning of neural networks
Author: Pavel Kasík
Department of: Theoretical Computer Science and Mathematical Logic
Supervisor: Mgr. Roman Neruda, CSc.,
Institute of Computer Science of the ASCR, v. v. i.
Supervisor's e-mail address: Roman.Neruda@mff.cuni.cz

Abstract: Designing neural networks topologies is s complicated problem when we consider general network structures. Evolutionary algorithm can provide us with interesting solutions of this problem. This work introduces an evolutionary algorithm for evolving neural networks. One of the possible algorithms for evolving neural networks is the NEAT algorithm. The goal of this work is to modify and enhance abilities of the NEAT algorithm. Improvements are focused on utilizing position of a neuron in network, improving crossover procedure and introducing solution of algorithm parallelization that preserve abilities of both NEAT and the new algorithm.

Keywords: NEAT, evolution, neural networks

Prologue

Artificial neural networks are very interesting computation model with universal approximation ability, but there is not much known about designing the structure of the network. There are plenty of ways how to connect neurons in the structure, types of neurons that can be combined, and even the possibility of approximation the biological model more precisely.

But the more complex the model is, the less we know about its abilities. The bigger the model is, the less efficient are learning algorithms and we cannot be sure if the designed topology is minimal and sufficient. Usually we can only use the trial and error method, and every trial can take plenty of time.

On the other hand evolving neural networks can provide us with unorthodox structures that can be smaller and more efficient than these designed by human. Of course, there are plenty of problems with evolving neural networks, too. The search space defined by all possible structures and values of weights is enormous. The most common way of reducing the search space is evolving only the weights, but this way we lose the possibility of creating unorthodox structures. Other problem is encoding the neural network into the genome, that can limit the evolution in finding possible solution. Ingenious encoding can reduce search space. A guiding fitness function is very desirable too. Even intelligent mutations make the evolution algorithm more efficient. Lack of these features will not stop the evolution, but it will make it practically unusable.

Aim of this work is to introduce a possibly new algorithm for evolution of neural network motivated by existing approaches. The algorithm should be able to evolve both parameters and topologies.

Structure of Diploma Thesis

This work is divided into several thematic chapters. The first chapter contains brief introduction into artificial neural networks and evolutionary algorithms. Main purpose of this chapter is to clarify basic notions of these two artificial intelligence branches. The second chapter introduces the NEAT algorithm, that was the main inspiration for this work. It contains detailed definitions and information about the algorithm that will be used lately throughout the work. The third chapter is devoted to ideas and motivations that were behind creating of the new algorithm presented in this work. The fourth chapter presents (multi-)agent structure of implementation and suggest possible solution of parallelization and synchronization. The fifth chapter contains detailed description of presented algorithm implementation with more technical aspect than previous chapters. Information from theoretical chapters three and four is utilized here without deeper analysis. The sixth chapter contains description of experiments with complete settings and results of the experiments. At the end in the seventh chapter, there is a conclusion of the whole work (i.e. its results) and possible future improvements or modifications of the presented algorithm.

Chapter 1

Artificial Neural Networks and Evolutionary Algorithm

In this chapter we summarize definitions used in the branch of artificial neural networks and outline problems connected with designing of networks. Second part of this chapter is dedicated to the evolution algorithm and basic definitions connected to it.

1.1 Artificial Neural Network

Artificial Neural Network or simply *neural network* is s biologically inspired computational model. It is a structure composed of simple computational units (neurons) connected together. There are many kinds of neural networks that differ in learning capability and suitability for different kind of problems. Generally the most common type are *feed-forward networks*, that have structure of direct acyclic graph. But there is not much known about how to design such general structures despite that we know how to learn them. A special case of feed-forward network is the (feed-forward) *multilayer network* which has the capability of universal approximation and for which there are some clues how to design it.

The general structure of feed-forward multilayer network is composed of several layers of neurons that are (fully) connected (see fig. 1.1). The first layer is an *input layer* (neurons receive input numbers as their own input), the last layer is an *output layer* (outputs of neurons are output of whole model) and all other layers are *hidden layers* (neurons receive inputs as the output of neurons from previous layer). Basic computation unit of neural

network is called *neuron*.

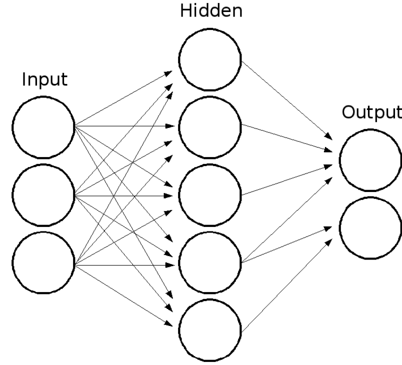

Figure 1.1: feed-forward multilayer neural network with 3 layers and fully connected input and hidden layer.

Every neuron is a simple computational unit (see fig. 1.2) that summarizes all inputs (weighted outputs of previous layer) subtracts *bias* and uses the result as the argument for an *activation function*. The result of the activation function is the output of the neuron. There are several kinds of activation functions, which in fact influence approximation capabilities of the whole neural network, but the most common is logistic sigmoid or hyperbolic tangent function (that both preserve universal approximation capability).

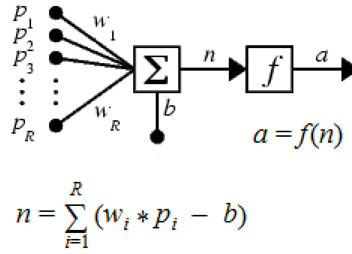

Figure 1.2: mathematical model of neuron, where p_i are inputs, w_i weights, b bias and f is an activation function applied on sum.

The abilities of feed-forward multilayer networks are relatively well explored, and moreover there is known algorithm for learning the network, i. e. setting the weights. On the other hand, there is not much known about

setting the number of layers and the number of neurons in each layer as well as about designing more general topologies. The problem with design of the neural network is not only a problem of feed-forward multilayer network, it is a more general problem.

The problem with efficiency of learning is another reasons for using evolution algorithm for constructing neural networks (not to mention that the biological model for artificial neural networks was designed by evolution too).

1.2 Evolutionary Algorithm

Evolution algorithm is a procedure for finding solutions of problems inspired by nature (which is most evident in terminology). The only requirements for using this technique is the ability to code solution of the problem in a sequence of genes and the possibility of construction of the *fitness function* that can evaluate the solution.

Every solution of the problem is called *specimen*. The algorithm operates not with one solution, but with a set of solutions (specimens) called *population*. Specimen is defined by list of *genes* that describes the solution of a problem.

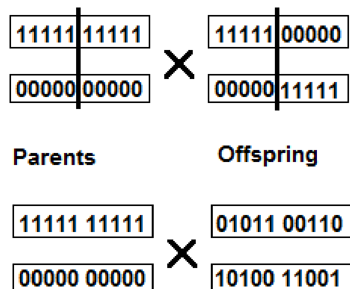

Figure 1.3: Example of *one-point crossover* in first case and *uniform crossover* in second case. in both cases *genes* are simply 0 or 1

The main procedure iteratively modifies the population an tries to select the best specimens for new population. The loop (see fig. 1.4 for diagram of evolutionary algorithm) of the evolutionary algorithm has three main steps:

1. mutation - mutations are applied on every specimen with some probability. They generally modify values in genes or add, remove whole genes.

2. evaluation - every specimen is evaluated by a fitness function, which determines how good is the solution of the problem (specimen).

3. creating new population - new population is created as a selection based on fitness from the old population. Genes of selected specimens are usually used for a crossover procedure, that from two (or more) specimens creates a new one (or more) offspring. Crossover procedure generally mixes sets of genes in order of creating new one. In case of linear sets of genes there is mostly used *one-point crossover* of two parents (see fig. 1.3) that cuts list of genes into two parts and switch one part with other parent. Similarly works the "multiple-point" crossover and the extreme is the *uniform crossover* that randomly chooses every gene from some parent (thus it can be used in case of more than two parents for example). These three steps are repeated as long as we are not satisfied with solution (i. e. the best specimen).

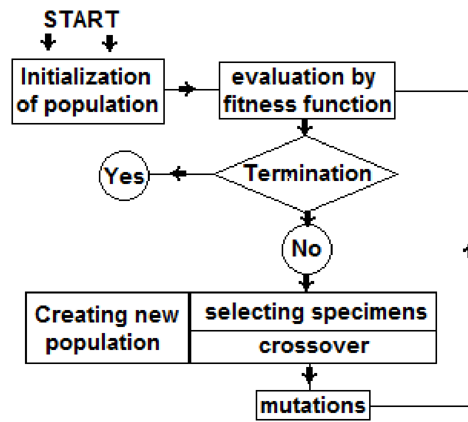

Figure 1.4: Diagram of evolutionary algorithm with three main steps: evaluation, mutation and creating new population. The main loop is continuing until termination conditions are met.

To supply some more detailed information, there is a theory about using the crossover procedure, which tells us that solution of the problem should be coded as a sum of features that are possible to encode as a subset of genes (of the whole solution). The closer the genes of the feature are to each other in the (linear) genome, the better is the chance that the feature survives

crossover and is combined with other desired feature. In other words, the solution of the whole problem should be a combination of desired compact blocks of genes, that itself provides us with relatively well evaluated solution. If the encoding of the problem does not meet this requirement, then the crossover is something like a multiple random mutation with hardly predictable effect and we can have rightfully some doubts about it. Thus in some cases the crossover procedure is not necessary (or even desired, for example in evolutionary programming [5]).

Chapter 2

Description and Analysis of NEAT

In this chapter the NEAT algorithm, that was the main inspiration of this work is described. Main features, definitions and analysis of its abilities are described (more deeper analysis of some key features is presented in next chapter).

2.1 NEAT

NeuroEvolution of Augmenting Topologies is an evolutionary algorithm which simultaneously optimizes both topologies and weights of neural network. It has two main features: indexing genes to allow simple crossover between topologies and sorting specimens into species to adjust fitness. Now we describe the whole algorithm step by steep (as an evolutionary algorithm) and introduce data structures necessary for its running.

The initial population contains individuals with minimal topologies (i. e. fully connected inputs and outputs without hidden nodes), and the evolutionary algorithm develops more complex topologies by adding new nodes and connections. Before we can describe how it is done (by *mutations*) we have to define the corresponding data structure (*genes*).

2.2 Genotype

Every specimen contains linear list of nodes and linear list of connections between nodes (for example see fig. 2.1). Every connection is defined by starting node (In), ending node (Out), weight, activity (enabled or disabled) and innovation number (Innov). Nodes are defined by their number and type (input, output, hidden).

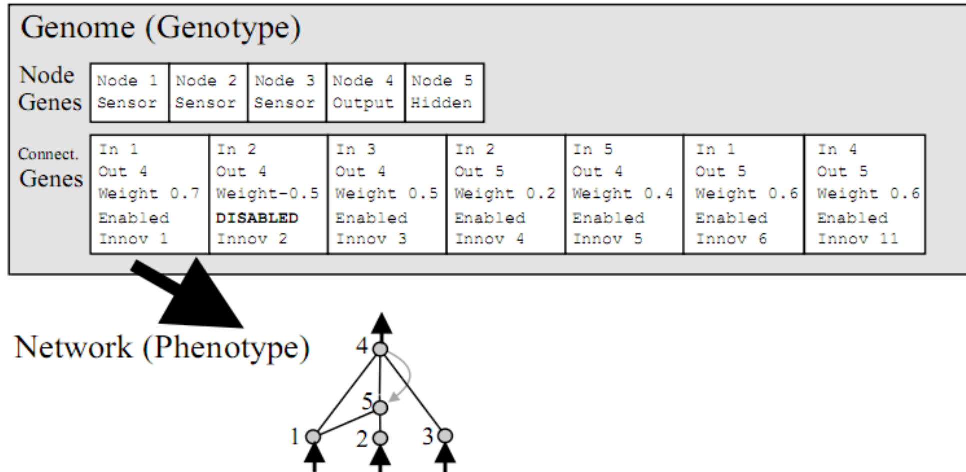

Figure 2.1: Example of simple neural network and its possible encoding into gene. As you can see, genes contain all necessary information to reconstruct the network. (figure used from [1])

2.3 Mutations and Innovation number

There are two types of structural mutations adding new connection and adding new node. Adding new connection means simple initiation of variables: In, Out, Weight, Enabled and innovation number (see fig. 2.1) and adding the whole gene into the list of genes. *Innovation number* is a global incremental index for naming genes, i. e. the way how to recognize identical genes (identical with respect to topology).

In every generation when new gene appears (structural mutation occurs) it is compared with other new genes in this generation. If there is a match (based on In and Out variables), the newly occurred gene receives the same

innovation number (same as all identical new genes in this generation). If there is no match, the new gene receives new innovation number, and the counter is incremented. Thus, we are able to recognize same mutations that occurs in different specimens in one generation (all identical genes in one generation have the same innovation number). The succession of innovation numbers represents the genealogy of specimen and allows us to crossover two specimens without thorough topological analysis.

Adding new node involves adding new record into the list of nodes, adding two new genes connecting new node with two other nodes and disabling connection between these two neighbours (if it exists). See figure 2.2 for easier understanding.

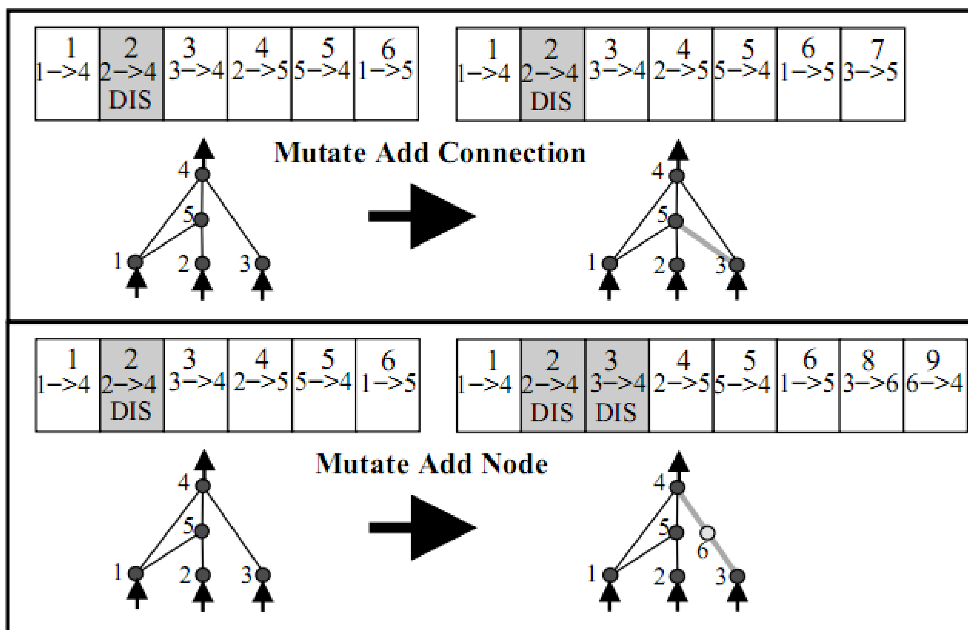

Figure 2.2: Example of two basic structural mutations. One adding new connection and other one adding node.(figure used from [1])

2.4 Crossover

Crossover of two specimens uses the idea of innovation numbers which provide us a simple approach how to recognize similar genes. Two linear lists of

parent's genes are sorted and compared. If both parents have genes with the same innovation number the offspring receives randomly one of each pair of matched genes. If the genes are only in one of the parents then they are called *excess* or *disjoint*. Disjoint genes are within the range of innovation numbers of both parents. Excess genes are genes with higher innovation number than all opposite parent's genes. Disjoint and excess genes are inherited from the more fit parent (see figure 2.3 for simple example).

2.5 Speciating and Fitness

Speciating is a method how to protect new topologies in population. First of all we need to classify all specimens into *species*. Every specimen is compared with list of species and classified (in case of a new species the new record is added). For species comparison we use following formula 2.1:

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \overline{W}, \tag{2.1}$$

where N is number of genes in larger genome, E is number of excess genes, D is number of disjoint genes, $\overline{W}$ is average wight difference of matching genes, and coefficients c_1, c_2 and c_3 allow us to adjust importance of every factor. Representative of species are randomly selected from previous generation. Every specimen is compared with these representatives and placed in the first species where distance δ is lower than threshold. Classification into species is used for adjusting the fitness of every specimen by formula 2.2:

$$f_i' = \frac{f_i}{\sum_{j=i}^{n} sh(\delta(i,j))}, \qquad sh(\delta(i,j)) = \begin{cases} 0 & \text{if } sh(\delta(i,j)) < \delta_t \\ 1 & \text{if } sh(\delta(i,j)) \geq \delta_t \end{cases} \tag{2.2}$$

where the numerator is the calculated fitness value and the denominator is count of specimen in species (the sharing function sh is set to 0 when distance δ is above threshold δ_t; otherwise, $sh(\delta(i,j))$ is set to 1, for more details see [4]). This fitness adjusting protects new and small species and reduces successful species to avoid overwhelming the whole population by one species. Motivation for this is the observation that a neural network with topological innovation has usually lower fitness than its ascendant and thus a lower chance to reproduce. Because of specieating a neural network with topological innovation has a chance to be sorted into different species than other offsprings and thus it can be favoured by fitness adjusting.

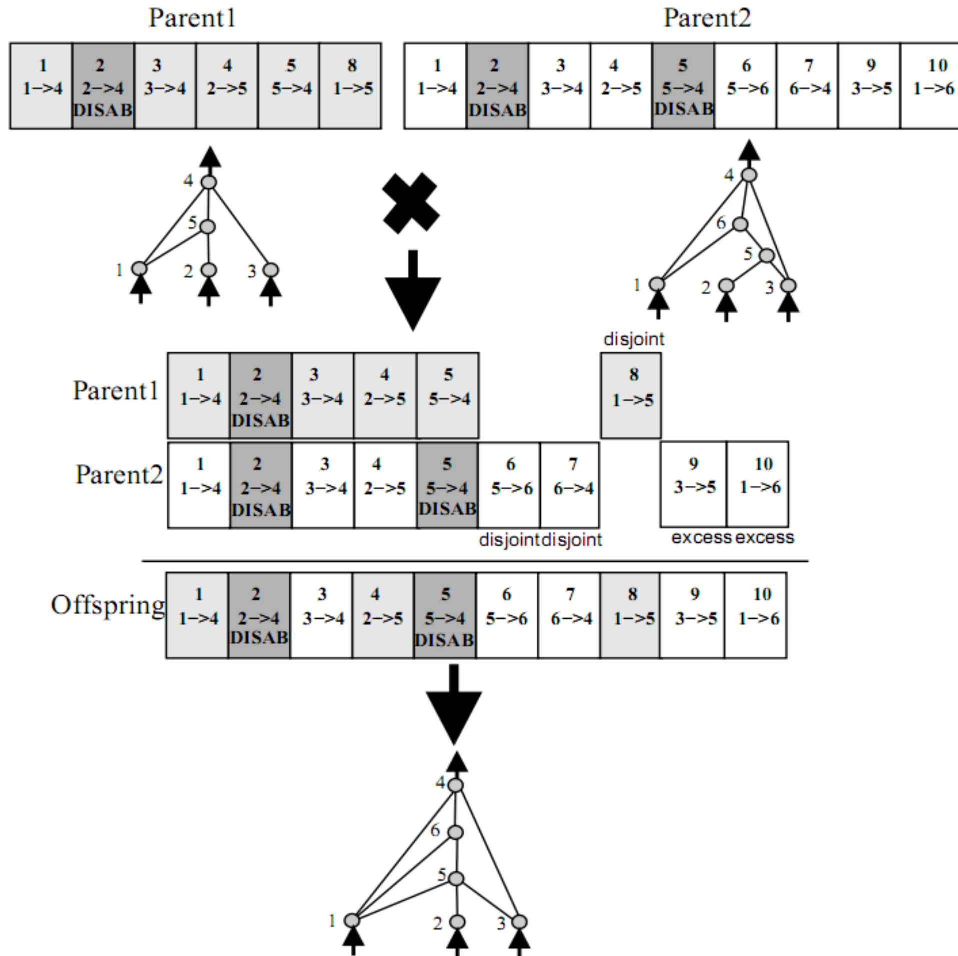

Figure 2.3: Example of crossover of two specimens (Parent1 and Parent2). Although topology of both parents look different, crossover is possible even without topological analysis. Matching genes (match based on *innovation number*) are inherited from random parent. Disjoint and excess genes are inherited from more fit parent – in this case, both parents have same fitness, thus all genes are inherited.(figure used form [1])

2.6 Incremental Growth

The last feature of NEAT is reducing the search space by starting from minimal structures. New topologies are introduced as structural mutations occur and survived selection because of their fitness. Thus, only the successful (i.

e. meaningful) topologies are preserved during the evolution. The algorithm itself tries to find minimal topology that maximizes the fitness function. More precisely, evolutionary algorithm can get stuck in local optimum and adjusting weights provide as bad result as adding new connection or node (generally structural mutation produces offspring with worse initial fitness compare to their parents). In that case it is possible that enlarging the network could help, and thus the evolutionary algorithm naturally produces specimens with new nodes or connections (because of inability to improve specimen by adjusting weights).

On the other hand, when it is possible to improve the fitness by adjusting the weights, the offspring has better chances to survive into the next population than offspring with structural mutations. The fitness adjusting mechanism prevents overwhelming the whole population by one specie, not reproducing the most successful specimens.

Technically there is also problem with terminating the evolutionary algorithm in the right time. Even when the topology is correct, it is possible that the improvement can be achieved only by adjusting few weights and for evolutionary algorithm it could be hard to find them (because generally mutations are random). Thus, the termination condition of algorithm could be satisfied by "good enough" solution. For example in case of binary classification problems the output 0.7 could be good enough and we will not insist on values very close to 1. In fact when we s have correct topology with weights close to optimum we can reach the optimum by teaching the network by some traditional gradient-based learning algorithm.

Chapter 3

New Algorithm

In this chapter there are summarized ideas that were behind the design of the new algorithm and there are described main differences from NEAT, i. e. naming of nodes to improve preserving structures by crossover, redesigning the crossover procedure, and few minor modifications at the end of the chapter.

3.1 Motivation

The intention of our work was to improve and modify NEAT in order to get a new algorithm with different features and better abilities. So we can start with analysis of NEAT's behaviour. First of all the crossover procedure is like a multiple mutation of weights in most cases, more precisely the offspring has the same topology as one of the parents. In case of equal fitness of both parents the result is unpredictable because of genes encoding (for example of NEAT crossover see fig. 2.3 or 3.1). As was told in the first chapter, the meaning of crossover should be the recombination of parents abilities to achieve offspring containing the best of both (all) parents. In the case of NEAT the crossover is more like (maybe smart) multiple mutation and the idea of genes recombination is fading away.

So if we presume that substructures (subgraphs) of neural networks have any meaning, and thus the crossover is something more than a multiple random mutation, there is not much chance that any complex substructure survive NEAT's crossover (in case of equal fitness). In most cases there is not even possible to inherit different structures form both parents. Lack of the possibility to inherit different structures from both parents is caused by

names of the nodes. The naming of nodes does not reflect position of the node in the structure of the neural network, and it does not allow to distinguish different structures. Of course there is a mechanism to distinguish genes (connections) thanks to innovation numbers but in most cases of crossover the fitness of both parents will be different and thus only one topology will be inherited (without any modification). In case the fitness of both parents is equal the crossover procedure can not preserve substructures in parents.

For example, it is possible (either naturally during evolution, or intentionally) to develop a partial solution that provides us with correct value on one output node in one specimen, and another specimen with correct value on output node (for example see fig. 3.1). Especially when different subset of inputs are necessary for correct value on each output node it is highly improbable that NEAT's crossover of these two specimens preserve abilities of both specimens. This is not a desirable behavior of a crossover.

The second observation focused on innovation numbers. It is a very simple and efficient way how to recognize similar genes, so why do we keep list of new mutations only during one generation? Especially in the beginning of the evolution it is highly possible that the same mutation occurs in next generations, but we will not be able to recognize it without the list of new mutations from previous generations. To improve similar genes recognition we propose a way how to name nodes with respect to topology and thus much more precisely recognize topologically identical mutations.

3.2 Node Naming

Our first modification is aimed at names of nodes. To be able to identify different substructures we decided to name new nodes by combination of two node's names between which we add a new one (for description by pseudo-code see fig. 3.2). To define it more precisely (for schematic example see fig. 3.3):

- Initial nodes of the network are named by numbers in brackets. If the network has n input neurons and m output neurons, then the names of input neurons will be "(1)"', "(2)",..., "(n)" and the names of the output neurons will be "($n+1$)", "($n+2$), ..., "(m)" (initial structure is the same as in NEAT, that means fully connected inputs and outputs without any hidden nodes).

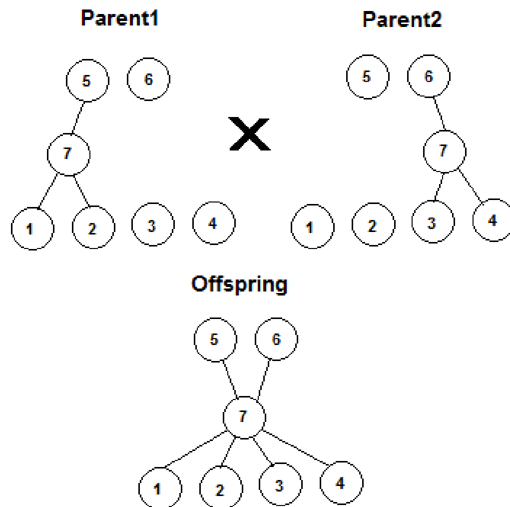

Figure 3.1: Example of NEAT's crossover. Presume that Parent1 has 1 hidden node, that calculates from inputs 1 and 2 good output 5 and Parent2 has 1 hidden node too, that calculates from inputs 3 and 4 good output 6 (in both cases hidden node has number 7). In NEAT's terminology all genes (used here) are excess or disjoint, and they will be inherited only from the more fit parent. Even if both parents have equal fitness the result of crossover is indicated in figure. We can doubt that at least one parent's ability is inherited (i.e. it is no big problem to construct model where this result of crossover is useless)

- Any new node (added by structural mutation) is named by the names of its neighbours (any new node is connected with two other nodes). Presume the name of the neighbours are (x), (y) and lexicographic order of these names is $x < y$. Then the name of the new node (added one) will be "$((x)(y))$".

- If node with constructed name already exist there is "+" added at the end of name (if + is already there is another one added and so on. . .).

Node's names are reflected into the names of connections too (In and Out variables) so we are able to better recognize similar genes.

This naming can not cover up all graph isomorphisms, but it is computationally effective (no topological analysis is necessary) and still it can recognize similar substructures in graphs.

Let **x**,**y** be two nodes between which we want to add new one **z** (x,y selected randomly for example). x.name and y.name are names of these two nodes.

Let x.name be lexicographic before y.name, i. e. x.name < y.name .

Set z.name to (x.name y.name): z.name="("+x.name+y.name+")" .

while exist node in list of nodes with name z.name do
z.name append + : z.name=z.name+"+"
endwhile

add **z** to list of nodes

Figure 3.2: Description of naming procedure by pseudo-code.

Other problems appear when trying to crossover structures with these names. Of course we are still able to find matching genes and correctly crossover two list of genes without more work than in original NEAT. But we have to pay attention to merging list of nodes because their are not identical like in original NEAT and we have to compare names of the nodes (not only position in list, which in NEAT reflect name too).

3.3 Crossover

If we want the ability to inherit substructures from both parents and have chance to inherit different features of both parents, we have to modify the crossover too. We decided to implement a *"merging crossover"* that merges booth list of nodes and list of genes (in case of genes pair with same innovation number choose randomly one).

This caused a problem not from the technical point of view but a theoretical one. With this merging crossover used on genes with new names it is very hard to develop small topologies. With more precisely differentiation of genes and nodes, the result of merging (offspring) is usually bigger than parents.

One way how to deal with this problem is to enhance structural mutations. If species are improving their fitness (there is better specimen in

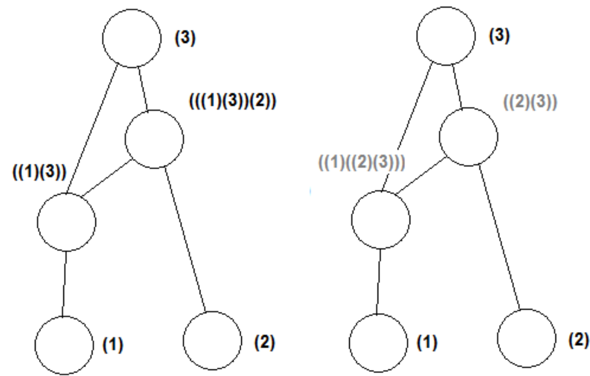

Figure 3.3: Two examples of nodes names and also an example of problem with graph isomorphism. Exact names of nodes depends on order of adding nodes. In left case there was firstly added node between nodes (1) and (3). After that there was added node between new node ((1)(3)) and node (2). At the end there was added connection between (((1)(3))(2)) and (3). In right side case there was firstly added node between (2) and (3). After that there was added node between new node ((2)(3)) and (1). At the end there was added connection between ((1)((2)(3))) and (3). It must be pointed out, that this is the worst case that can happend and this topology can be created by several other ways with better similarity.

species than in previous generation) the structural mutations occur only with some probability (it was for example 50 per cent in our tests). This mean that without consideration of individual probabilities of mutations, there is a chance that structural mutations will be skipped. Mutations variating parameters (weights) are not affected. This mechanism give chance to optimize weights without adding new nodes and connections into topology.

Another way how to deal with the crossover problem is to implement different crossover. Together with merging crossover, a classical one point crossover was added. The result of this new crossover procedure are three offspring - one as a result of merging crossover and two as a result off one point crossover.

It is not ideal because of linear list of genes, which does not respect the topology and thus the crossover could create isolated subgraphs with no connection to the rest of the network. To improve the situation there is a cleaning procedure which removes nodes that are in different graph component than inputs and outputs. It finds components of connectivity

within only one pass of genes list, and then it removes all redundant genes and nodes.

It is computationally comparable with crossover procedure in NEAT and the other possibility is to leave isolated subgraphs there and hope that some structural mutation connect them back to the rest of network. But this option enlarge the search space, because of leaving (maybe temporally) redundant genes in genotype (list of genes).

Computationally, the new crossover procedure is slightly more complicated than original NEAT crossover because of the cleaning procedure, but it allows to produce offspring with comparable genome's list length (even shorter than parents) and still there is some chance to preserve at least part of the abilities of both parents.

These two kinds of crossover preserve the advantages of merging (i. e. chance for inheriting good features form both parents) and improve exploration abilities of the whole evolutionary algorithm because of one point crossover.

A positive result of these modifications could be the possibility to evolve of only subset of required outputs and than finish learning with modified fitness to achieve fair values on all outputs (incremental evolution). On the other hand we cannot anticipate minimal possible topology (in cases of incremental evolution), because all restrictions were unknown from the beginning, but we can find a suitable solution of problems thats are too hard without some decomposition. This behaviour is a result of preserving substructures which can be counterproductive – two features of network could be calculated by (separated) structures instead of one (more complicated).

3.4 Minor Modifications

There are also some other minor changes in our algorithm in comparison to NEAT. One of them is the modifycation of formula for calculation distance between specimens to the formula 3.1:

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \overline{W} + \frac{c_4 O}{M} + \frac{c_5 T}{M} \tag{3.1}$$

where two fractions was added. The first one is $\frac{c_4 O}{m}$, where O is number of nodes with different names (names that are not in opposite specimen), c_4 is coefficient to adjust wight of this difference and M is number of nodes in larger specimen. This allow us to make bigger difference between networks

with relatively low count of connections and different nodes. The original NEAT is not able to recognize the position of nodes in structure and thus can not distinguish them (nodes) in different specimens. Without this fraction formula is still able to recognize different structures via different genes, but this allow us to give bigger (or less) importance to different nodes. The last fraction $\frac{c_5 T}{M}$ from formula 3.1 is for making difference between nodes with different activation function. M is number of neurons, c_5 is coefficient for adjusting the importance, and T is number nodes with different activation function (difference is counted only if exist in opposite specimen neuron with same name). Of course, this is useful only when heterogeneous networks are evolved.

Other minor modifications with respect to NEAT contain enhancing probability of structural mutation by possibility to skip them in case of improving fitness of the whole specie (defined above in this chapter in section devoted crossover), and persistent list of genes to improve identifying similar genes. If large amount of genes is anticipated (large network evolved) it is possible to set the maximum length of the genes list to improve computational demandingness.

Chapter 4

Agent Parallel Implementation

In this chapter, the solution of our evolutionary algorithm is outlined. The synchronization between agents is also analyzed and our solution presented. The agent implementation was chosen for possible application in multi-agent system.

4.1 JADE

Java Agent DEvelopment Framework (or simply JADE) is a framework fully programed in JAVA for the purpose of simple multi-agent system development [3]. It contains a middle-ware that complies with FIPA specification ([7]) and a set of graphical tools that supports debugging and deployment of agents.

One of the advantages of implementation in JADE is the possibility to divide the work between multiple agents. JADE provide us with simple way of communication between agents with communication system. Moreover, planing behaviors inside the agent is not preemptive, and the agent is typically executed in one thread. Because of this we have an absolute control over agents behavior and there are no problems with data consistency in one agent. We only have to decide the way of problem decomposition, data exchange and synchronization.

4.2 Agent's structure design

Parallelization of evolutionary algorithm could be done in several more or less difficult ways. The outcome of this effort will be improvement of the algorithm performance, because the work can be divided to multiple processor units. Two main approaches to parallelization of evolutionary algorithms are distribute only computation of fitness, or having several populations with sharing some specimens [6]. In this case the model with multiple populations is used.

The agent's structure is designed as a radial with multiple *computation agents* performing evolutionary algorithm and one *synchronization agent* in the center for the purpose of synchronization and data sharing. In order to communicate with the rest of the multi-agent system, a *communication agent* connected with the center was designed to ensure distribution of task and data. Since the connection to the multi-agent system was not part of the problem, this agent has not been implemented yet (for scheme see fig. 4.1).

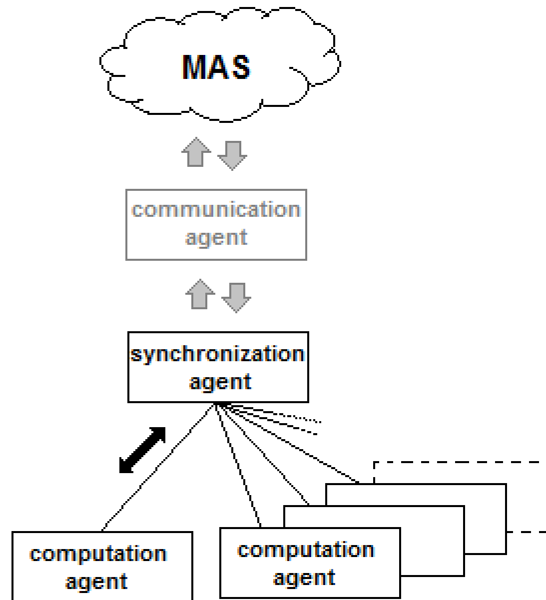

Figure 4.1: Scheme of agents communication and task decomposition.

The computation agent performs evolutionary algorithm as it is described in chapter above (the detailed technical description will be provided in the

next chapter). The computation agent communicates only with the synchronization agent, while the synchronization agent ensures sharing of specimens necessary for the purpose of algorithm (without sharing of specimens the algorithm will not be distributed).

Synchronization agent keeps list of all mutations to be able to synchronize innovation numbers across all computation agents and to send them back the corrected numbers. Besides this, the agent keeps list of all species of all computation agents to be able to determinate where to send the specimens in order to keep species together. Thus, the last function of the synchronization agent is to distribute incoming specimens among computation agents. If no computation agent has a similar specie, one agent is selected by round robin rule and the specimen is send to it.

The communication agent is not implemented yet as it was mentioned above. The purpose of this agent will be communication with multi-agent system, receiving problems, distributing them to the rest of agents and providing solutions to contractors.

4.3 Synchronization

If we allow transferring of specimen between agents we have to sort out a problem with synchronization of innovation numbers. Without it the transferred specimen will not be able to crossover correctly with other specimen. If we want to be able to crossover any two specimen we have to ensure at some point that both specimen will have their innovation numbers synchronized. Thus there has to be some central authority with list of all mutations to be able to synchronize any two specimen.

This is the reason why we decided for simple radial structure of agents. The central synchronization agent keeps a list of all mutations and it can provide correct innovation number for any genom. It could be possible to synchronize only incoming specimens with the list of mutations of receiving computation agent, but the amount of transferring agents (and all genes that they have inside) would have to be smaller than amount of new mutations. Another possibility could be the synchronization of transferring agents inside receiving computation agent, but the synchronization agent needs synchronized genes in order to be able to correctly count distance between specimens (to determine the specie).

To summarize it, synchronizing genes in receiving computation agent is not possible with this model of transferring specimens. We could transfer for

example only the best specimens (or the worst – it depends on the chosen model [6]) but we would have to leave the idea of keeping species together. Synchronizing the whole agents (with list of receivers genes) in synchronization agent is computationally time consuming and it could be profitable only in case of small amount of transferring agents. But the small amount of transferring agents means small amount of new species and this is most probably a sign of bad setting of the algorithm coefficients (in the formula calculating distance between specimens). Selected solution suits for idea of keeping species together. The synchronization agent could be a bottleneck of the whole system, but as long as it is able to synchronize at least one generation of new genes (even if it deletes previous generation), the result should be comparable with NEAT.

4.4 Algorithm Parallelization

All computational agents are equal and independent on the rest of the system if we do not consider transferring specimen. The content of communication with synchronization agent is a list of species, a list of new genes for synchronization of innovation numbers and a list of specimens designated to transport to another computation agent.

For the purpose of specimen transferring we decided to keep all population synchronized. At the end of mutation (see fig. 5.2) the list of all new mutations is send to the synchronization agent. By the reply all new genes are corrected with new innovation numbers. The start of the communication is timed before the fitness calculation, because for the purpose of the fitness the synchronization is not necessary. The second aspect of the timing is presumption that fitness function takes most of the computation time and thus there is some space for delay evoked by communication. If the time is not enough, the progress of the algorithm inside one computation agent is stopped and the agent waits for message with correct innovation numbers.

After the synchronization of innovation numbers the list of representatives is send to synchronization agent. Besides, the list of specimen of low count species is sent too. The idea is to keep species together and not distribute them among more agents. It is an attempt to ensure the diversity and living space for more species and to collect specimen of low count species together to improve their chances for survive. One specimen of specie in a whole population can survive, but it is highly probable, that the result of crossover with specimen of different specie will not be same specie.

Chapter 5

Description of New Algorithm Implementation

This chapter contains a ideas and mechanisms mentioned in previous chapters. All of them are put together into a detailed description of new algorithm with all its implementation aspects.

5.1 Genotype

Encoded genes are the mostly same as in the case of NEAT. From technical reasons, In and Out variables are still numbers, i. e. position of node in the list of nodes. In our implementation variables In and Out are called *To* and *From* (we find it more descriptive). To be able to determine a similarity of genes based on names the of nodes, every gene contains its name. The name of the gene is a composition of names of nodes that it connects. More precisely it is a string in a form "name of *From* node"-"name of *To* node" (see fig. 5.1).

The list of nodes was slightly modified: every node contains information about its *Name*, *Bias* value, number of *Inputs* connections and *Type*. The Type it is not an input/output/hidden value, but it is the type of activation function. Activation functions are predefined, and Type contains only an index of corresponding activation function (for example see fig. 5.1). Number of inputs and outputs is defined by the problem, that network solves and it is not a variable that could differ in every specimen. Thus the first nodes in the list of nodes are inputs, after them there are outputs (both numbers are fixed for every problem) and after them there are hidden nodes. Number

of inputs of every node was stored for network consistency tests on several occasions. During calculation of network output, the node with most known inputs is calculated next – in case of acyclic graphs there is always a node with all known inputs, in case of cyclic graph we have to do a topological analysis to break the cycle or use a heuristic and calculate output of node with most known inputs. Because the topological analysis is time consuming, the heuristic has been preferred in our case.

Genome (Genotype)

Nodes:

Name: (1)	Name: (2)	Name: (3)	Name: ((1)(3))	Name: (((1)(3))(2))
Bias: 0.7	Bias: 0.3	Bias: 0.4	Bias: 0.1	Bias: 1.9
Inputs: 0	Inputs: 0	Inputs: 1	Inputs: 3	Inputs: 1
Type: 2	Type: 2	Type: 2	Type: 2	Type: 2

Connections:

From: 0	From: 1	From: 0	From: 3	From: 1	From: 4	From: 1
To: 2	To: 2	To: 3	To: 2	To: 4	To: 3	To: 3
Weight: 0.7	Weight: 0.3	Weight: -0.3	Weight: 1.3	Weight: 1.1	Weight: 1.3	Weight: -1.1
DISABLED	DISABLED	ENABLED	ENABLED	ENABLED	ENABLED	ENABLED
Innov: 1	Innov: 2	Innov: 5	Innov: 6	Innov: 9	Innov: 10	Innov: 15
Name: (1)-(3)	Name: (2)-(3)	Name: (1)-((1)(3))	Name: ((1)(3))-(3)	Name: (2)-(((1)((1)(3)))	Name: (((1)((1)(3)))-((1)(3))	Name: (2)-((1)(3))

Network (Phenotype)

(3)
((1)(3))
(((1)(3))(2))
(1)
(2)

Figure 5.1: Example of network and its encoding into the list of genes and list of nodes. First two nodes in the list of nodes are input neurons, third one is output and all others are hidden nodes.

5.2 Mutations

Probability of all mutations is normalized to fixed length of a genome, we refer to this normalized value as *mutation rate*. With longer list of genes the probability of mutation is getting lower. For example the probability of modifying weight of connection is normalized to 10 genes. Thus setting mutation rate to 0.6 means that statistically there will be modified 6 weights no matter of length of genome list (therefore there is a good meaning in setting mutation rate higher than 1 for larger networks). This allow us fast exploration of search space in the beginning of algorithm and more precise tuning weights in later phases of algorithm.

For all experiments we have used only general mutations and not problem based ones (i. e. special mutations for specific problem). There are two structural mutations: adding node and adding connection. Both of them are enhanced by skipping possibility in case of improving species. Both of them do not create connections leading from outputs and to inputs. It is in order to improve search abilities of algorithm in the beginning.

As for mutations changing parameters there are biased mutation (value := value + random_gaussian_number) and unbiased mutation (value := random_gaussian_number) modifying values of connections and biases. There are two more mutations, one enabling and disabling connections (genes remain in the list, so it is not a structural mutation), and one changing type of node activation function.

5.3 Selection and Crossover

To improve abilities of evolutionary algorithm there were implemented mechanism of *elite* that preserve into new generation a preset amount of the best specimen of previous generation. Another one similar mechanism is the *hall of fame* that protect presets amount of best specimens of the whole evolution and adds their copies into the population before crossover. Last mechanism influencing selection is used from NEAT, it is checking if species are getting better. If there is no better specimen produced in preset amount of generation, the fitness of every member of specie is set to zero. Selection procedure for crossover is simple roulette method that distributed the probability of selection according to the proportion of fitness in population.

For the purpose of crossover there are two specimen selected and three offspring created. One offspring is a result of merging crossover, and two are result of one-point crossover. During the crossover procedure s cleaning procedure as was described in chapter 3 is used. The size of population is not fixed and it can temporarily get bigger than preset. It is because of incoming specimen, that are added into the population. When new population (new generation) is created, the size of it is nearest possible to preset number (crossover produces three offsprings so it is the nearest multiple of three larger than the preset size).

5.4 Computation Agent

Computation agent performing evolutionary algorithm has two behaviours (procedures). *Evolutionary behaviour* performs the evolution, and *Communication behaviour* that services incoming communication. For better understanding how computation agent works see fig. 5.2.

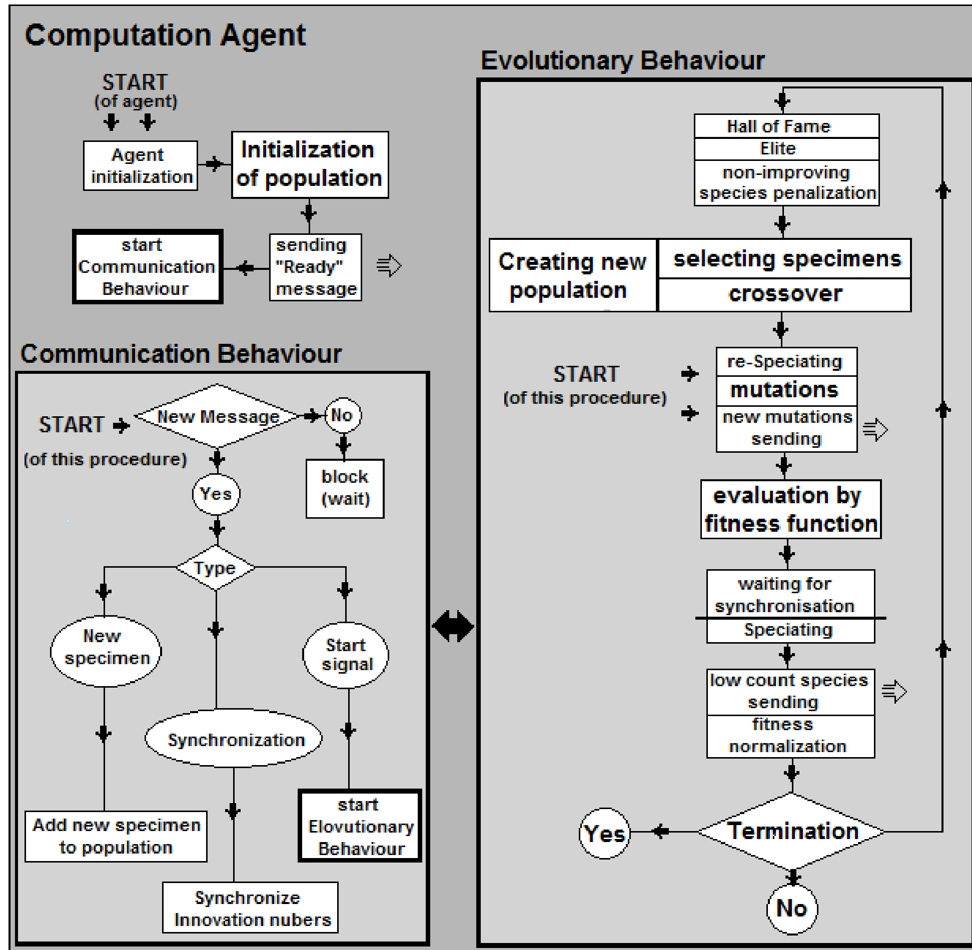

Figure 5.2: Scheme of computation agent.

First, the agent initializes all internal variables, population and sends "ready" signal to synchronization agent. At the end of the initialization communication behaviour starts. Communication behaviour is waiting for message from synchronization agent (thread is blocked until a message ar-

rives). When signal for start arrives, communication behaviour starts evolutionary behaviour. Both behaviours are active and they can switch after the other procedure makes a step (i. e. the behaviour finishes its *action()* function). Communication behaviour has only one step (i.e all incoming messages are executed), and evolutionary behaviour has six steps. All six steps (of evolutionary behaviour) are depicted as boxes in fig. 5.2.

Evolutionary behaviour starts from mutations. "Re-Speciating" tries to determine if a specie is improving (in the beginning it does not make sense but the algorithm is working in a loop). To determinate it a specimen is classified into a specie. This is necessary to be able to determine if we should apply probability to skip structural mutations (to get network a chance to improve weights if the specie is improving - an idea of this is described in chapter 3). If there is no suitable specie, a skipping feature can not be applied and all mutations are regularly calculated. If the specie is improving, there is 50 per cent chance to skip structural mutations. After finishing the mutations, the list of new mutations is send to synchronization agent.

The next step of the algorithm is evaluating every specimen via fitness function. During this step a message from synchronization agent with corrected innovation numbers should arrive. At the end of the step a communication behaviour gets chance to process messages. If the synchronization message has a delay, the next step holds the progress of evolutionary behaviour, and the agent has to wait for synchronization message.

Another step after synchronization is speciating of all specimens into species. If there is no suitable specie for some specimen, a new representative is created to represent new specie. The next step sends to synchronization agent low count species and normalize fitness as prescribed by formula 2.2.

Another step (on fig. 5.2 it is on the top of the diagram) actualizes the Hall of Fame (HoF contains best specimens ever), the elite is determined (the best specimens of population) and the best specimen of every specie gets its place in new population too (i. e. it receives flag to be preserved to new population).

After this, the crossover procedure takes place and the new generation is created. The loop continues as long as termination condition is satisfied (the most common condition is a maximum number of generation).

5.5 Synchronization Agent

Synchronization agent is simple having only one behaviour that reacts on incoming messages. The agent has a list of all mutations, list of all computation agents and list of representatives of all computation agents. Computation agents register to synchronization agents via "Ready" message. When the synchronization agent receives a message with genes, it tries to find them in the list of all genes and correct their innovation number or add them into the list (and correct the innovation number too if it is necessary). When the synchronization agent receives message with specimens, it tries to find agent with similar specie. If synch. agent can not find suitable computation agent it (synch. agent) choose one of the computation agents by round robin rule and send it a specimen (representative is added to the target's representatives list too).

When the synchronization agent receives list of representatives, it replaces an old one. It is possible that there is an inconsistency (a computation agent can send list of representatives before it received new specimens), but it is not a big problem, and it is a price for being able to execute most of the algorithm without a synchronization signal. If we would like to insist on absolute synchronization, it could happen that the system would be as fast as the slowest agent. In any case, there will be more synchronization signals necessary and they will be burdening the synchronization agent. So this one flaw in synchronization has been taken as an acceptable price for better performance of whole system.

Chapter 6

Experiments

This chapter contains several results of our new algorithm and comparison with the NEAT algorithm. There were selected three types of experiments: XOR experiment to verify basic functionality and for comparison with NEAT, Building dataset as a representative of approximation problem and Cancer dataset as a representative of classification problem. Both datasets are form Proben1 set of benchmarks for neural networks [2].

6.1 XOR Experiment

The XOR experiment has been selected because it is not a linearly separable problem, and it requires hidden nodes in the network. This should verify the ability to create such hidden nodes and do not stuck in local optimum. Another reason for choosing this problem is the possibility to compare the results with original NEAT's results described in source [1].

For this problem the initial population has been set to 100 specimens. Probability of adding edge has been set to 0.4 (normalized to 30 genes, i. e. with more than 30 genes the probability is linearly lowered). Probability of disabling edge has been set to 0.2 (normalized to 30 genes too). Probability of adding node has been set to 0.05 (normalized to 10 nodes). Probability of connection value perturbation has been set to 0,6 and probability of bias value perturbation has been set to 0.3. Probability of reseting value of connection has been set to 0.2. Probability of reseting value of bias has been set to 0.1 and probability of deactivating the edge has been set to 0.2 (all probabilities normalized to 10 genes/nodes). Probability of changing type of activation function has been set to 0, because it was not desired to change

	ø number of neurons	ø number of genes	ø output value
kill specie 20	6.4	10.5	0.838 / -0.856
kill specie 10	5.6	8.9	0.888 / -0.870
NEAT	4.35	7.48	unknown

Table 6.1: Table with results of XOR experiment. Caption *kill specie* means number of generation, after which the penalization is applied for non-improving species. Row captioned by *NEAT* shows result of the original NEAT presented in [1]. The column captioned by *ø number of genes* shows the average numbers of non-disabled genes.

it. As an activation function the hyperbolic tangent has been chosen. Numbers for perturbation has been chosen from (normal) Gaussian distribution divided by 10. Numbers for reset has been chosen from (normal) Gaussian distribution. Size of elite has been set to 1 (i.e. the best one of every specie has been preserved by crossover) and size of hall of fame has been set to 3 (i. e. the three best specimens of whole evolution has been add to population before crossover). Coefficients of specie distance formula 3.1 has been set to c_1=5, c_2=5, c_3=10, c_4=8 and c_5 has no meaning, because the types of activation function have not been changed during the evolution. The threshold δ has been set to 1. Numbers of generations during which the species do not have to improve (without penalization) has been set to 20 (in the first half of tests) and to 10 (in second half of tests). The termination condition of the evolutionary algorithm has been set to 100 generations. There were 20 tests together and the fitness function has been $(1 \pm x)^2$ and after x reached ± 0.4 there were added another bonus +2 (i.e. it was consider as a good classification).

The evolutionary algorithm successfully found satisfying solution in all 20 tests. As you can see from table 6.1 the results are comparable with the original NEAT algorithm despite that there were expected problem with evolving very small topologies (because of more precise distinction between genes and nodes). To see comparison in the right view it is necessary to point out several things. First of all, in the source material ([1]) there is not specified what solution (output value) has been accepted as a correct. Because it is a classification problem the values (for example) 0,3 and 0,7 (in case of sigmoidal activation function) could be considered as a correct classification. Thus the evolution can be stopped sooner and thus lower risk

of adding unnecessary connections or even nodes. In the original paper there is also stated, that average number of generations was 33 (in the worst case 90). In our case there were always 100 generations (because there were no other termination condition). The average achieved output values are stated in table. Secondly, because of different network encoding into genes, in our case the number of genes should be higher (to be able to correctly compare these numbers) because in the list of genes there are no biases (every bias value is in node object). Finally, the attempt to repeat the results of NEAT was only partially successful. In source material there is no description of select procedure and more detailed description of mutations (exact numbers and description missing). Our best possible reconstruction of the original NEAT algorithm has similar results (number of genes and nodes) as described, but the results were not very stable and it takes twice more generations to achieve them. With small improvements (for example improve NEAT by enhanced mutations used in new algorithm) the results were much more better, but we can not be sure if it was due to the mutations, because of lack of detailed description of some parts of algorithm.

The overall result of this experiment is, that the new algorithm has no problem with this simple task and it is able to develop relatively small topology comparable with original NEAT, despite the slightly worse results than expected.

6.2 Building Experiment

The building experiment is an example of hard approximation (prediction) problem. It uses the building dataset from the proben1 benchmark tests. The dataset contains originally 6 inputs values (date, time of day, outside temperature, outside air humidity, solar radiation and wind speed) and 3 output values (consumption of electrical energy, hot water and cold water in building). The 6 inputs values are re-encoded into 14 inputs and both inputs and outputs are normalized to range $< 0; 1 >$. The dataset contains 4208 examples all together.

The first setting (*setting 1*) for this experiment was: 50 specimens in population, 100 generations, 0.6 probability of connection adding, 0.1 probability of node adding, 0.9 probability of connection perturbation, 0.4 probability of bias perturbation, 0.3 probability of connection value reset, 0.13 probability of bias value reset and 0.3 probability of connection disabling. Normalizations, size of elite, hall of fame and mutations (choosing new value) were

the same as in the XOR experiment. Type of activation function was the hyperbolic tangent in hidden and input nodes. In output nodes there were the sigmoidal activation function (because of output values normalization). Coefficients of specie distance formula 3.1 has been set to c_1=6, c_2=6, c_3=8, c_4=10. The threshold δ has been set to 1.5. Maximum size of the list of all genes (for the purpose of synchronization) has been set to 1000 genes. The fitness for one output was $(1 - |x_0 - x|)^2$, where x_0 was target value and x was output value. Gain on all three outputs and of all examples were summarized to get fitness of the specimen.

The low count of specimens and the low count of population for the first try has been set because of time demandingness of the whole evolution (4208 examples and starting topology containing 17 nodes plus 32 connections). From same reason (speeding the algorithm) the maximum number of remembered genes was set.

The second setting (*setting 2*) was almost the same but instead of single population of 50 specimens, there were 3 populations (three computation agents) of 25 specimens (75 together). Species with less than 3 specimens were send for sharing. Purpose of this settings was the comparison of the results of single-population evolution and multi-populations evolution with similar number of specimens. The disadvantage of multi-populations model is the inability to crossover any two specimens. On the other hand, the advantage is that it is less time consuming.

The third setting (*setting 3*) was multi-populational too, and there was 50 specimens in each of three populations (150 specimens together). Maximum number of generations has been set to 150. Species with less than 5 specimens were send for sharing. Purpose of this settings was to show the progress of convergence with better settings.

All three settings were repeated 10 times and an average square errors and standard deviations were counted. Results of the experiments are in table 6.2.

In all 3 tests more than 75 per cent of examples were approximated with less than 1 per cent square error (i.e. less than 0.1 error in absolute numbers). This means the algorithm, in limited time and with limited resources, was able find relatively fair solution. An interesting observation concerns the number of used connections. In all experiments during the crossover procedure, there were 25 per cent chance, that the inherited gene will be re-enabled even if it is disabled in both parents (and the inherited gene was always enabled if at least one parent has this gene enabled too). In spite of

	ø neurons	ø genes	ø error	ø std	1%	4%
setting 1	20.5	16.7	0.043	0.014	9508	409
setting 2	18.7	17.9	0.049	0.011	9217	571
setting 3	24.4	17.3	0.037	0.015	9898	275

Table 6.2: Table of building experiment results: *ø neurons* is average number of neurons, *ø genes* is average number of enabled connections, *ø error* is average square error, *ø std* is standard deviation (of 10 runs) of square errors, *1%* is average number of outuput values with less than 1 per cent sqaure error and *4%* is average number of output values with worse than 4 per cent square error. Number of all output values is $3 * 4208 = 12624$ together.

this half of the genes in the best specimens was disabled.

The results can be compared with results in Proben1 [2], that were achieved with traditional gradient-based learning algorithm RPROP (fast backpropagation variant similar in spirit to Quickprop) on several topologies. The topology with no hidden nodes achieved during (average) 400 learning epochs mean 0.78 of squared test set error percentage (in our test with settings 3: 3.7 per cent). Several topologies with hidden nodes achieved mean between 0.76 and 1.70 of squared test set error percentage. Number of epoch were between 300 and 2600 (depends on topology).

6.3 Cancer Experiment

Cancer experiment is a classification problem from Proben1. The dataset contains 699 examples defined by 9 input values and 1 output value. From the sixth input characteristic 16 values is missing and were defined as an average value. The output has 2 values defining 2 types of tumour (benign/malignant). All input values were normalized to range $< 0; 1 >$.

Two models were used. The first model (*model 1*) with one output neuron (i. e. state of the neuron defines the kind of tumour) and second model (*model 2*) with two output neuron – for every kind one output. For both models same settings were used: 100 specimens in population, 200 generations, 0.6 probability of connection adding, 0.1 probability of node adding, 0.9 probability of connection perturbation, 0.3 probability of bias perturbation, 0.3 probability of connection value reset, 0.1 probability of bias value reset and 0.15 probability of connection disabling. Normalizations, size of

	ø neurons	ø genes	ø errors	ø std
model 1	26.1	35.5	19.7	1.73
model 2	22.9	37.4	51.2	4.66

Table 6.3: Table of cancer experiment - *model 1* with one output and *model 2* with two outputs:*ø neurons* is average number of neurons, *ø genes* is average number of enabled connections, *ø error* is number of wrong classifications, *ø std* is standard deviation (of 10 runs).

elite, hall of fame and mutations (choosing new value) were all the same as in the XOR experiment. The type of activation function was the hyperbolic tangent. Coefficients of specie distance formula 3.1 has been set to c_1=5, c_2=5, c_3=8, c_4=9. The threshold δ has been set to 3. Maximum size of the list of all genes (for the purpose of synchronization) has been set to 1000 genes. The fitness for one output was $\pm sgn(x)x^2$, and for the values above 0.4 (or under -0.4) there were as a bonus of +1. In model with 2 outputs they were summed together and one more additional bonus for fitness was applied: $699 - 2e$, where e is number of examples, where both outputs give wrong value. Output values above 0.4 (or under -0.4, respectively) have been considered as correct classification.. Results of the experiments are in table 6.3.

In case of model 1 networks recognized more than 97 per cent of samples. In case of model 2, the success rate was lower (93 per cent). To compare with Peoben1(i.e. with gradient-based learning algorithm RPROP) – linear networks have average 20.5 wrong classifications (97 per cent success rate), and (different) multilayer networks have 8.0, 10.2, 9.6 wrong classifications (98.9, 98.5, 98.6 success rate).

Chapter 7

Conclusion

This chapter contains brief recapitulation of the whole work and several ideas for future work.

7.1 Recapitulation

The intention of this work was to introduce an algorithm for artificial neural networks evolution. We have developed a neuroevolution learning procedure capable of training both topologies and weights of the network. The new algorithm utilizes ideas of the NEAT algorithm and tries to further improve them. The idea of keeping track of topologies evolution by means of innovation numbers has been improved by introducing a more sofisticated node coding. This approach is able to detect larger class of equivalent topologies throughout the evolution run. Furthermore improved crossover and mutation operators have been introduced. And finally, the algorithm has been implemented in a distributed way by means of the JADE agent framework and tested on several benchmark datasets. Sadly, the direct comparison with the NEAT algorithm failed – not because of our new algorithm, but because of the inability to reproduce the performance of NEAT referred to in the literature. As an alternative we provide a comparison with traditional (advanced) gradient-based algorithms. Despite it is known that neuroevolutionary techniques searching much more complex space usually need an order of magnitude more time to achieve comparable errors than gradient methods performing local search of weight parameters only. The datasets are public and can be used to indirect comparison of introduced algorithm with any possible evolutionary algorithm.

My personal gain form this work was a chance to create and test interesting algorithm enhanced by my own ideas and familiarization with practical aspects of agent design, experienc with using JADE and JAVA.

7.2 Future work

In the future work we would like to improve our algorithm by modification of the list of genes. Instead of linear representation some tree structure could be very interesting. Maybe some tree structure will be necessary for the list of nodes too. The idea of this representation is to be able to crossover two topologies to merge or switch their substructures without additional analysis of connectivity. This idea come only recently after experiences with the original algorithm proposal and we did not have time to figure out useful tree representation.

The implementation of the algorithm is far from finished. It is fully operational, but every experiment has to be set manually and there is no user-friendly interface. There are several reason for it, and most cardinal is absence of application environment definition (i. e. the multi-agent environment and communication definitions were not a part of this thesis).

Bibliography

[1] Kenneth O. Stanley, Risto Miikkulainen (2002). Evolving Neural Networks through Augmenting Topologies. *Evolutionary Computation*, 10(2):99-127, The MIT Press, Austin, Texas, USA.

[2] Lutz Prechelt (1994). PROBEN1 - A Set of Neural Network Benchmark Problems and Benchmarking Rules. *Technical report 21/94*, Universität Karlsruhe.

[3] Fabio Bellifemine, Giovanni Caire, Dominic Greenwood (2004).*Developing Multi-Agent Systems with JADE*, John Wiley & Sons, Ltd.

[4] William M. Spears (1995). Speciation Using Tag Bits. In *Handbook of Evolutionary Computation.* IOP Publishing Ltd. and Oxford University Press, Oxford, UK.

[5] Lawrence J. Fogel (1999), *Intelligence through Simulated Evolution : Forty Years of Evolutionary Programming*, John Wiley.

[6] Erick Cantú-Paz, David E. Goldberg (1999) On the Scalability of Parallel Genetic Algorithms. *Evolutionary Computation*, 7(4):429-449, The MIT Press, Austin, Texas, USA.

[7] Foundation for Intelligent Physical Agents, website: *http://www.fipa.org*