Yevgeniy Ivanchenko

ADAPTATION OF NEURAL NETS FOR RESOURCE DISCOVERY PROBLEM IN DYNAMIC AND DISTRIBUTED P2P ENVIRONMENT

Master's thesis

Mobile computing

17/08/2004

University of Jyväskylä

Department of Mathematical Information Technology

Author: Yevgeniy Ivanchenko

Contact Information: E-mail: yeivanch@cc.jyu.fi

Title: Adaptation of neural nets for resource discovery problem in dynamic and distributed P2P environment.

Work: Master's Thesis

Pages: 99

Study Line: Mobile Computing

Department: University of Jyväskylä, Department of Mathematical Information Technology

Keywords: Peer-to-Peer networks, resource discovery, neural networks, Multi Layer Perceptron, Self Organizing Maps

Abstract: Peer-to-Peer networks are complex systems that have widely spread recently and thus providing efficient management to them is necessary and at the same time challenging task. The aims of this work are: discovering of decision mechanism of NeuroSearch algorithm developed in the Agora Center, testing NeuroSearch under dynamic conditions and finding good solution for optimization problem. These three goals are lying on the middle part of the path to solving the main problem: developing new approach to train the core mechanism of NeuroSearch, which is based on the MLP, under dynamic conditions and in supervised manner.

Acknowledgements

The Thesis was written at the Department of Mathematical Information Technology and Agora Center, University of Jyväskylä, Finland. I would like to thank them for presenting me that great opportunity. I would like to express my gratitude to my supervisors Jarkko Vuori and Mikko Vapa for their help and support during the writing the Thesis.

Finally I would like to thank Helen Kaikova and Vagan Terziyan for their help, support and invaluable advices during my studying and living in Finland.

University of Jyväskylä, 30.7.2004

Abbreviations

BFS	Breadth First Search
BMU	Best Matching Unit
BP	Back Propagation
DHT	Distributed Hash Table
EA	Evolutionary Algorithm
EC	Evolutionary Computing
GA	Genetic Algorithm
HDS	High Degree Search
LDS	Low Degree Search
MLP	Multi Layer Perceptron
NAM	Network Animator
NN	Neural Network
P2P	Peer-to-Peer
RBA	Rule Based Algorithm
SOM	Self-Organizing Map
TTL	Time To Live

Contents

1 Introduction

1.1 Background

In recent years, interest to Peer-To-Peer (P2P) technology has increased noticeably, especially due to P2P file sharing systems such as Kazaa [Kazaa], eDonkey [eDonkey], Gnutella [Gnutella] etc. However, P2P is not only about file sharing, it is also about establishing multimedia communication networks based on P2P concepts or resource sharing [Schollmeier 2001]. Nowadays there are many applications of systems that use P2P paradigm, for instance, such as collaboration systems [Groove], computation systems [SETI@Home] and infrastructure systems [Edutella]. Also P2P systems will be able to be used for interaction between ubiquitous devices. P2P systems are based on the dynamic and distributed architecture, which allows guaranteeing robustness, scalability, efficient use of available resources, adaptation and self-organization.

P2P can be defined as system consisting of peer processes that can act as server and client at the same time. In pure P2P system there is no centralized point. A peer can initiate requests, and it can respond to requests from other peers in the network. Users have a higher degree of autonomy and control over the services they utilize.

Now let us turn clock back and look through history of appearance of Peer-to-Peer. Some people think that P2P paradigm is something new, but it is not entirely so. Early Internet was organized in such a way that all hosts were equals. And they had some characteristics, which can characterize any distributed system such as P2P. Each host was able to perform routing (locate machines), accept FTP connections (file sharing) and accept telnet connections (distributed computation) [Oram 2001]. After that, when needs of users increased and at the same time power of computers still was not so big, Client-Server architecture spread in the Internet and Peer-to-Peer temporary passed from the scene, awaiting its chance to appear again in a new role. That chance was given to P2P with appearance of Napster, when power of modern computers increased to satisfy high requirements of users. Although several years ago an activity of Napster was stopped (of course now Napster still exists, but people have to pay for the music), because of copyright

problems, appearance of Napster was initial point to resurrect lost interest to problems, which are related to P2P networking.

1.2 Research Problem Statement

It is difficult task to provide efficient methods to handle P2P networking, because of absence of an infrastructure in pure P2P models. But in particular these kinds of networks are of interests in most researches. One of the most important tasks that should be solved is organization of an efficient search in P2P networks. Nowadays a lot of methods have been proposed in this area. But none of them uses neural network solutions except the NeuroSearch algorithm [Vapa et al. 2004] developed in the Agora Center.

1.3 Contribution of the Thesis

Since in the Thesis, a lot of methods from different scientific domains are used. Therefore it might be difficult to understand how different parts of the Thesis are related to each other. In this section brief introduction to the problems, which I want to solve on the Thesis is given.

The contribution of the Thesis may be seen from two different positions. First it aims to solve problems related to adaptation of decision mechanism of NeuroSearch for dynamic environment. Second the Thesis investigates NeuroSearch trained with help of Evolutionary Computing to look inside of decision mechanism. Having solved the above described problems we will be able to provide efficient solution for optimization of NeuroSearch in dynamic environment. It is because results obtained from simulation under dynamic conditions of NeuroSearch that was trained in static environment can answer in general about the feasibility of using such algorithm in dynamic environment. Investigation of NeuroSearch that was trained with the help of Evolutionary Computing is done with help of Self-Organizing Maps (SOM) [Kohonen 2001]. This investigation is needed to know more about the inner processes in NeuroSearch. Therefore this investigation can help answering existing questions about NeuroSearch such as which inputs are needed, which

inputs are responsible for particular decision and so on. This can help to use more efficient training strategies in future works.

1.4 Cheese Factory Project

The thesis was made within Cheese Factory project [Cheese], which is going in University of Jyväskylä at Agora Center. The project studies P2P communication and behaviour of P2P networks concentrating on distributed search of resources and their efficient use. In the project Chedar P2P platform has been developed. Chedar is used as distributed computing and storage system for distributed applications. To study behaviour of P2P network P2PStudio emulator environment has been developed as well. P2P Studio can control the whole Chedar servent network using graphical user interface. One of the researching goals of Chedar and P2PStudio is realtime resource location. Currently Chedar uses three searching algorithms: adaptive flooded search, link rating/selection and NeuroSearch. My research is entirely dedicated to NeuroSearch algorithm.

1.5 Structure of the Thesis

This work is dedicated to resource discovery problem, which is one of the main problems in P2P networking. The thesis consists of eight chapters. Chapter number two reviews P2P architectures, their properties and existing search strategies in P2P systems. Chapter number three introduces the used neural network architectures (Multi Layer Perceptron and Self Organizing Maps) and some methods for preliminary determination of neural network architectures. Also this chapter describes Matlab environment for simulation of neural networks. Chapter number four is entirely dedicated to NeuroSearch that was developed in the Agora Center as part of Cheese Factory project [Cheese] and evolutionary computing. In chapter number five we introduce data analyzing techniques and apply them to know more about behaviour and decisions making mechanism of NeuroSearch. In chapter number six we consider aspects of P2P simulation, available simulation software and stages of development of P2P simulation for NS-2 [NS-2]. Chapter number seven presents evaluation of results obtained from simulation of Rule Based algorithm (RBA) and their comparison with results of Breadth-First Search algorithms in dynamic environment. RBA

is entirely based on NeuroSearch. It was produced by converting decision mechanism of NeuroSearch to set of rules using SOM. Also chapter number seven presents some optimization problems that are related to provision of efficient training in supervised manner. The thesis is concluded in chapter number eight.

2 Peer-to-Peer systems

This chapter reviews existing P2P systems, their characteristic properties and presented solutions in these systems for resource discovery problem.

2.1 Definition of Resource Discovery Problem

Resource discovery problem is illustrated in figure 2.1.

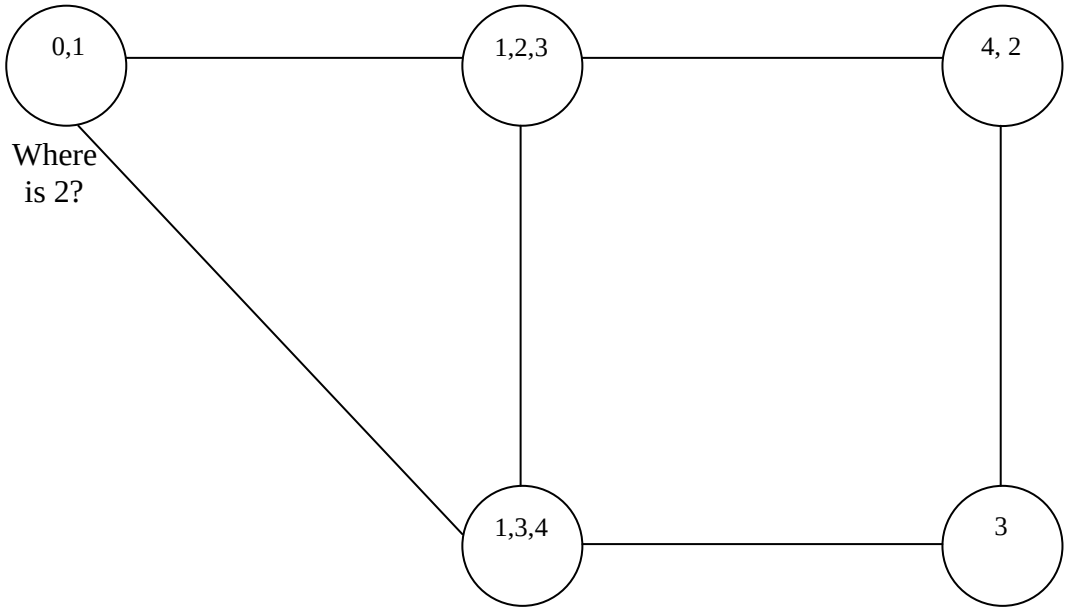

Figure 2.1 – Resource discovery problem

In figure 2.1 one can see that the network can be represented as a graph. Each node in the graph contains different amount of resources. Thus resource discovery problem comes to finding optimal searching strategies on the graph. In contrast to usual searching methods on the graphs (where we should find the shortest path between some two nodes), in resource discovery problem we should apply good search strategy, which uses efficient spread of the queries through the network to find satisfied amount of resources.

2.2 Peer-to-Peer Architectures

If we consider P2P networks in terms of traditional OSI model then we can notice that P2P interaction belongs to application layer. Development of such networks comes to building of logical interaction between peers. For example, in well-known P2P network as Gnutella neighboring peers might not be connected physically and some peer can be located in USA and its neighbor somewhere in Brazil.

Mainly P2P systems are subdivided into two models:

- Pure (Decentralized) P2P model
- Hybrid (Centralized) P2P model

Classical example of hybrid P2P model is Napster.

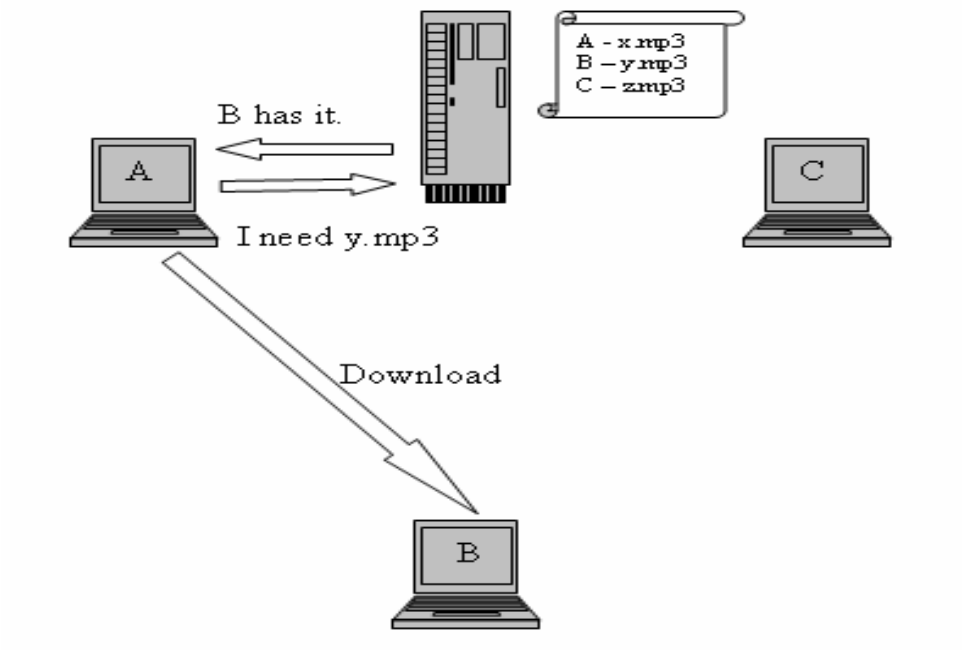

Figure 2.2 – Hybrid P2P architecture

The hybrid model has one central server to maintain transactions between all peers in the network. The server contains list of resources that each user has. When user needs to find some resource, he sends query to the central server and server sends back to user

information that matched his query. Figure 2.2 shows interaction between peers and server in the hybrid P2P model.

The pure P2P models such as Gnutella are characterized by absence of any centralized points. Each peer in such system is autonomous and can act both as server and as client. Thus pure P2P model completely satisfies this P2P definition. Figure 2.3 shows interaction between users in the pure P2P environment.

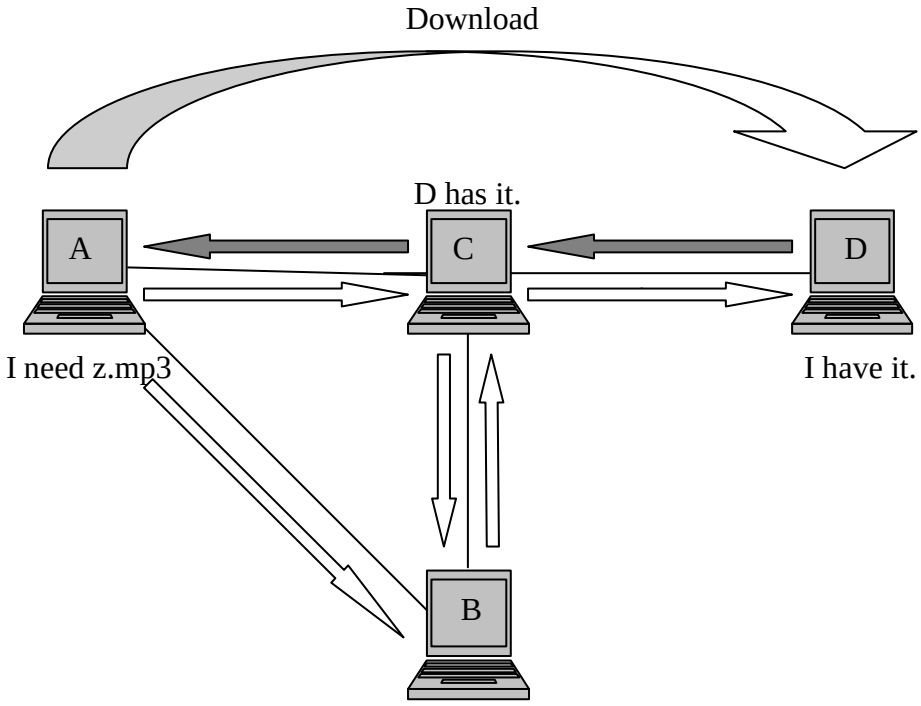

Figure 2.3 – Pure P2P architecture

Decentralized architecture is subdivided into structured and unstructured systems. The main difference between these two systems is that in structured systems we know beforehand location of resources or some hints how to find resources, in unstructured systems resources are distributed randomly.

Organizing of an effective searching mechanism in the hybrid P2P models is easier than in the pure P2P models. But the hybrid model has smaller reliability in contrast to the pure model because it has central server and if something happens with the central server then it will influence on the performance of the whole system. Efficient maintaining of the pure

P2P system is complex task, because of lack of an infrastructure. Providing solutions to maintain this system is a challengeable and at the same time a necessary task. Thus our research is dedicated to the pure P2P systems.

2.3 Small World and Power-law Networks

Recent studies [Iamnitchi et al. 2002] showed that P2P systems have characteristics of small world networks. Small world networks have two major characteristic properties:

- Small average path length – even if size of the network is large, there is a short path between any two nodes.
- Large clustering coefficient – the networks have relatively big connectivity between at least some nodes.

There are a lot of examples of such networks. For instance the World Wide Web [Albert & Barabasi 2001], where the nodes are web pages and the edges are the hyperlinks. Another example of small world networks is the Internet [Albert & Barabasi 2001], where the nodes represent routers or computers and the edges are represented by connections between them.

In the small world networks, typical distance between any two nodes scales as the logarithm of the number of nodes. Clustering coefficient of the one node can be determined using the formula (1).

$$C_i = \frac{2E_i}{k_i(k_i - 1)} \tag{1}$$

Where k_i is the number of edges of particular node *i* that would connect it to k_i other nodes in case of full connected graph, E_i is the number of edges that actually exist between them. To determine clustering coefficient of whole network we need to calculate average value of all C_i.

Naturally, all nodes in the network do not have the same number of links. A lot of networks including WWW [Albert & Barabasi 2001], Internet [Faloutsos et al. 1999], P2P [Ripeanu 2002] have power-law distribution of nodes' degree *k* (2).

$$P(k) \sim k^{-\gamma} \tag{2}$$

Where γ is parameter that depends on the type of the network, . These three key properties (shortest path length, clustering coefficient and power-law dependency) of P2P systems play significant role in studying and modelling of such systems. For instance, if we decided to generate P2P scenario based only on power-law distribution we could get some case when we have 'chain' of nodes that are connected maximum only with two neighbors increasing the average path length parameter. Thus it is very important to take into account all of these three parameters.

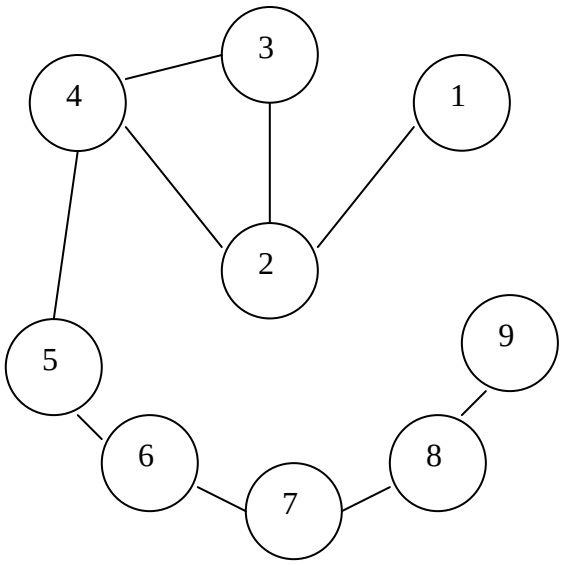

Figure 2.4 - Power-law graph with small clustering coefficient and large average path length

Figure 2.4 shows power law graph where we have chain from the node five to the node nine, this chain decreases clustering coefficient and increases average path length.

2.4 Peer-to-Peer Search Algorithms

Search in nondeterministic environment such as P2P environment is very complex task because we do not know exact location of resources and topology is always changing. Thus even using of very efficient deterministic search strategies like A* algorithm [Russell & Norvig 1995] is not possible.

Nowadays a lot of methods were proposed for searching in P2P networks. Most common search strategy is to use the Breadth First Search (BFS) algorithm, which was first introduced and tested in Gnutella network. This algorithm uses simple flooding mechanism to locate resources in the network. When user needs to find some resource in the network he just sends query to all neighboring nodes, which propagate this query to their neighbors except the node from which query was received and so on. If the query has been received earlier it is dropped. When query finds needed resource at some node, this node sends back reply to notify about this. Each query contains Time-To-Live (TTL) field that defines limitation on travelling path. BFS algorithm is foolproof method to locate all resources in the network, but we should pay high price for this, because the algorithm is not scalable.

To decrease flooding a lot of techniques were proposed. It has been shown [Lv et al. 2002] [Adamic et al. 2002] [Ata et al. 2003] that forwarding queries to only some neighbors can significantly decrease flooding of messages in the network and at the same time keep satisfactory level of located resources. One method is concluded in forwarding messages to a randomly chosen neighbor or to a randomly chosen group of neighbors (Random Walk). Another approach is concluded in selecting of the highest degree node (High Degree Search) or the lowest degree node (Low Degree Search) for further forwarding. All these methods act in depth-first search manner and thus have large latency.

Depth-first search algorithm with a limited depth size is used in Freenet [Freenet] network. Each node forwards the query only to one neighbor and after that waits for reply. If the query was satisfied it sends results to initiator of the query and if it was not it sends the query to another neighbor. Major disadvantages of this approach is that we should define not only depth limit, but also waiting time for satisfaction, which can vary for peers with different network connection.

Yang and Garcia-Molina [Yang & Garcia-Molina 2002] proposed three different strategies for searching in P2P environment. Their experiments were made with real Gnutella client. First method, which is called Iterative deepening is based on BFS algorithm. It uses some parameters, which should be determined in advance. First, it needs system-wide policy that determines depth levels of iterations, in other words this policy is just predefined set of TTL values for each iteration. If the query was not satisfied with first element from system-wide policy it resends the query with second element and so on. Also here needs be defined a waiting time of satisfaction. This parameter is similar to parameter, which is used in Freenet network. Second method, Directed BFS forwards queries to subset of neighboring nodes, which are selected with help of different heuristics that are stored locally on each node. These heuristics include number of replies returned for recent queries, the lowest average path for satisfying previous queries, number of sent messages and the shortest message queue. Third algorithm, Local indices is some kind of Iterative deepening. The difference between them lays in policy. Policy in Local indices specifies nodes at which the query should be processed.

2.5 Discussion

Methods that were mentioned in section 2.4 have some advantages and disadvantages. Using of one concrete method should be selected reasoning from starting requirements of each task. If it is more important only to find all instances of resources and paying attention to other criteria is not so important then we should use simple algorithms like BFS. If number of packets is crucial part of task and locating only part of resources' instances satisfies us then we should use more intelligent methods. P2P networks usually are characterized as large size networks with small-world properties thus using simple flooding techniques is not good solution for such networks.

Some from above mentioned algorithms require preliminary determining of some parameters, especially this is related to methods that were proposed by Yang and Garcia-Molina. Determining of such parameters can be difficult task, because P2P is unstable system and thus these parameters can vary for different networks and for different network's behaviour.

Nowadays degree based forwarding methods are quite efficient strategies for resource discovery problem in P2P network. It was shown [Ata et al. 2003] that High Degree Search can decrease the search delay compared to the Random Walk, and Low Degree Search can improve robustness against peer failure.

Using of Random Walk method is also quite efficient strategy, but it is difficult to evaluate performance of this algorithm because of random nature of this algorithm.

3 Neural Networks

Artificial neural networks (NN) have quite a lot similar characteristics with biological neural structures. They consist of elemental computing units (neurons), which are organized as complex structure with help of special connectors called synapses. NN can be applied to the areas where traditional methods do not work well or do not work at all. For instance, NNs are often used in such task as pattern recognition, approximation of functions, compression of images, forecasting and many others.

This chapter describes basic principles of NN computing. Without knowing ‘sources’ it is not possible to understand the problem completely. In other words this helps to understand why one concrete method or NN architecture can be selected and others not to solve particular problem. Information that is given in this chapter can be also used in future works to provide supervised training for resource discovery problem.

3.1 Multi Layer Perceptron

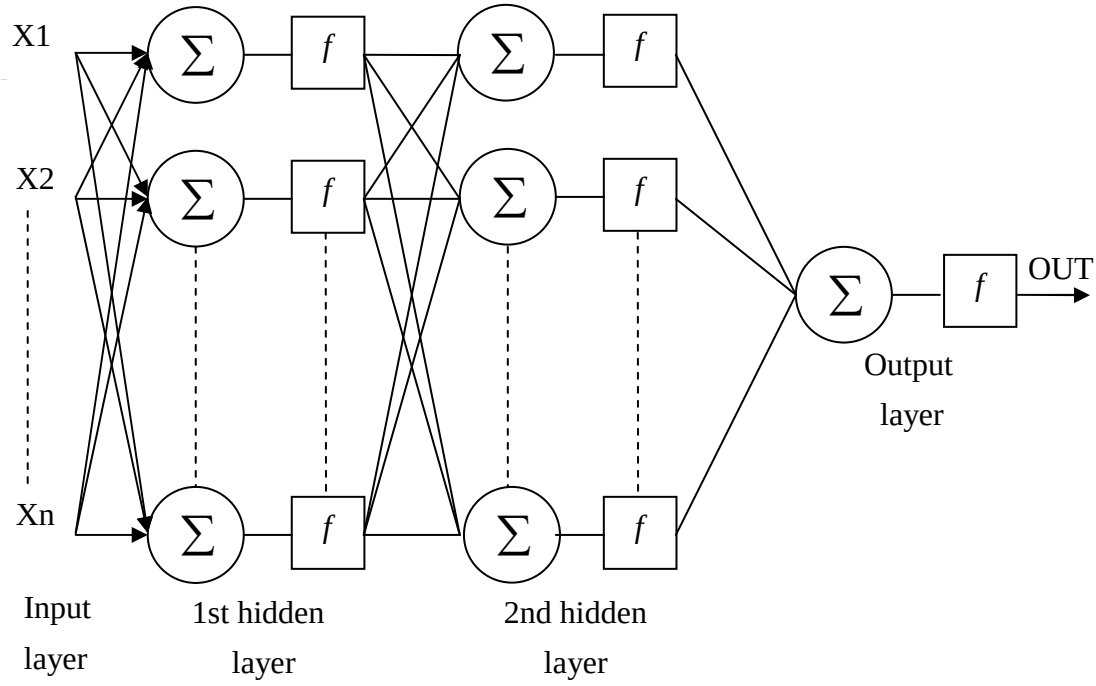

Figure 3.1 – Multi Layer Perceptron

Full-connected Multi Layer Perceptron (MLP) is shown in figure 3.1. Another name for MLP is feed-forward network. In the figure one can notice that circles are denoted as neurons, squares are denoted as their activation functions and lines are denoted synapses and axons. Axons go out from neuron and synapses go into neuron. Each synapse has its weight and function of each neuron is similar to ordinary weighted summator. Each neuron has its activation function. There are a lot of kinds of activation functions. The one of the first introduced activation functions was threshold activation functions (3).

$$F(x) = \begin{cases} 1, & x > 0 \\ 0, & otherwise \end{cases} \tag{3}$$

Nowadays most used functions are sigmoid (4) and hyperbolic tangent (5).

$$F(x) = \frac{1}{1+e^{x}} \tag{4}$$

$$F(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{5}$$

Activation functions can be viewed as amplification of artificial neuron. Activation functions like sigmoid and hyperbolic tangent can solve problem with noise satiation. The main point in this problem is how NN can process weak and strength signals at the same time. Weak signal needs high amplification and at the same time using strength signal with high amplification can lead to satiation of NN. Another purpose of activation function is to scale output of each neuron between 0 and 1. Applying of activation functions has one general requirement: these functions must have their first derivative in all points.

Usually MLP contains one input layer, several hidden layers and one output layer. The purpose of input layer is to forward signal to all neurons on the hidden layer. After that all neurons on hidden layer calculate their outputs and forward them further. Single neuron can solve only line-separable task, for example it cannot solve XOR problem. Illustration of XOR problem is shown in figure 3.2.

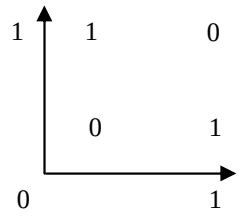

Figure 3.2 – XOR problem

In the figure 3.2 one can see it is impossible to use single line to separate clearly two clusters (ones and zeros). But two neurons united in NN will be able to solve such task.

3.1.1 Training of the MLP

There are a lot of strategies to train NN. Nowadays mainly two approaches are used:

- Supervised Learning
- Unsupervised Learning

One of the first introduced approaches was Hebb's method. This unsupervised method was introduced in 1949 [Hebb 1949] and the idea is the following: increasing of weight of each synapse is made according to correlation between two neurons, which this synapse connects. Mathematical interpretation of this rule is defined by formula (6).

$$w_{ij}(t+1) = w_{ij}(t) + NET_i NET_j \tag{6}$$

In formula (6) $w_{ij}(t)$ and $w_{ij}(t+1)$ denote weights of synapse between two neurons in time *t* and *t+1* respectively, NET_i and NET_j denote outputs of the first and the second neurons respectively. Further investigation [McEliece et al. 1987] of the Hebb's rule showed limitation of using this method with some patterns.

After introduction of Hebb's method, a lot of supervised methods were proposed, but all these methods were without straight theoretical background. Introducing the most powerful method called Error Back Propagation (BP) [Bishop 1995] played significant role in neural

computing. BP has excellent mathematical background. BP algorithm has the following steps:

1. The outputs of all neurons in the hidden and output layers are calculated using formula (7).

$$OUT_j = F(\sum_i w_{ij} x_i) \tag{7}$$

Where F is sigmoid activation function, w_{ij} is weight of synapse that connects ith neuron from previous layer to jth neuron at next layer, x_i is the output signal from ith neuron.

2. Error of MLP is calculated using formula (8).

$$E = \frac{1}{2}(OUT(x) - G(x))^2 \tag{8}$$

OUT(x) and *G(x)* are current and desired outputs respectively.

Updating of weights is defined by formula (9).

$$w_{ij}(t+1) = w_{ij}(t) - \alpha \frac{\partial E}{\partial w_{ij}} \tag{9}$$

On this step all calculations are made for the last (output) layer thus j is index of neuron from the output layer and i is index of neuron from the previous hidden layer.

The error function does not contain a dependence on weight w_{ij} thus implicit derivation of composite function are used:

$$\frac{\partial E}{\partial OUT} = \delta_j = (OUT(x) - G(x)) \tag{10}$$

$$\frac{\partial E}{\partial x_j} = \frac{\partial E}{\partial OUT}\frac{\partial OUT}{\partial x_j} = \delta_j OUT(1-OUT) \tag{11}$$

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial OUT}\frac{\partial OUT}{\partial x_j}\frac{\partial x_j}{\partial w_{ij}} = \delta_j OUT(1-OUT)OUT_i \tag{12}$$

3. On this step all calculations are made for the hidden layers. Calculations of derivations are made with the same formulas as in step 2, but formulas for finding δ_j are changed a little.

$$\frac{\partial E}{\partial OUT} = \delta_j = \sum \frac{\partial E}{\partial x_j}\frac{\partial x_j}{\partial OUT} = \sum \delta_j OUT(1-OUT)w_{ij} \tag{13}$$

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial OUT}\frac{\partial OUT}{\partial x_j}\frac{\partial x_j}{\partial w_{ij}} = (\sum \delta_j OUT(1-OUT)w_{ij})(OUT(1-OUT)x_i) \tag{14}$$

   To calculate δ_j error back propagation method was used. Partial derivatives can be found using only variables from the next layer. Weights of hidden layer are changed using the above presented formulas. If NN has several hidden layers then BP method is used in series for each hidden layer starting from the last hidden layer.

4. If converging conditions (usually it is small training error) is not met then return to step number one.

One can see from steps number two and three that training process is merely solving of error optimization task using gradient method. The meaning of BP method consists of using weighted sum of errors from next layer to the previous layer. Parameter α from (9) is used to define speed of learning process. Usually this parameter is quite small to guarantee convergence of the method. To determine this parameter compromise between speed and good convergence should be found.

After introduction of original BP method, a lot of modifications were introduced. Originally BP was used with random selection of one sample on each iteration.

Convergence time of this version of BP method was very slow. To decrease time of convergence batch method [Bishop 1995] was introduced. This method instead of one random sample uses whole training dataset on each iteration. Weight's updating is performed by using average value of error for the whole dataset.

Other approaches use adjustment of the speed parameter α. If training is slow then the speed parameter should be increased. If training is too fast then the speed parameter should be decreased. There are both local and global methods for adaptation of the speed parameter. The difference between them is that in global methods all weights have one common speed parameter and in local methods each weight has its own speed parameter.

The most interesting method is Rprop [Bishop 1995]. In this method value of derivative is not important. This method takes into account only sign of derivative. Convergence of this method is very fast. There are also a lot of methods that use high derivatives, for example such as Levemberg-Marquadt method [Bishop 1995].

3.1.2 The Overlearning Problem

After completing training process it is possible that NN merely learnt all samples from training dataset and at the same time it has poor possibilities to predict behavior of the system in the intermediate points. Classical example of the overlearning problem is shown in figure 3.3.

To avoid overlearning a lot of methods have been proposed. The simplest solution is interrupting the training process. But using of this method is dangerous because we can interrupt training in the point where NN did not reach quite good optimal point. Instead of this we could use test dataset together with training dataset. Interrupting of the process should be done in the point where difference between error on training and test datasets is minimal and at the same time error on training dataset is quite small. Figure 3.4 illustrates use of this method. One can see point A that is quite good moment to interrupt training.

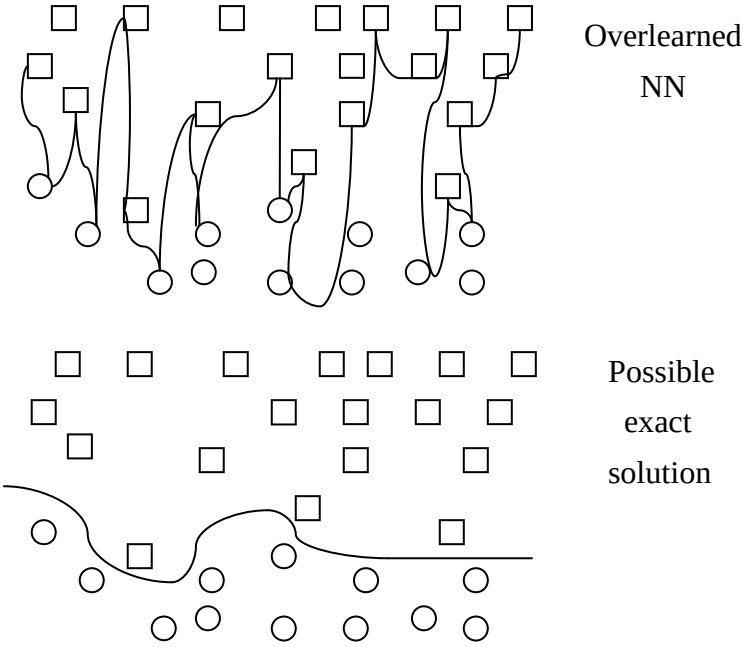

Figure 3.3 – The Overlearning problem

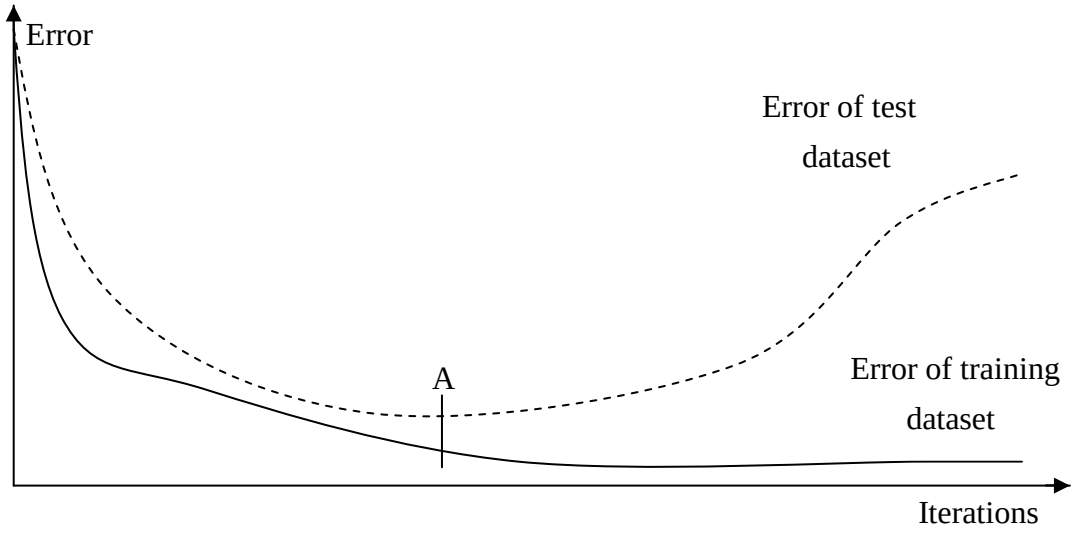

Figure 3.4 – Using of test dataset

Another good approach is to use penalty terms. Weight decay [Bishop 1995] is well known example of such methods. The penalty term in weight decay penalizes large weights. It is

quite easy to apply such method by adding new component ($-2\lambda w_{ij}$) in the weight update rule. The parameter λ is used to define strength of penalization. Recent researches [Krogh & Hertz 1995] show that weight decay can significantly improve generalization abilities of NN.

3.1.3 Projection of the MLP

Creation of MLP is quite complex task. A lot of experiments should be done before selecting the structure of NN. Despite on this some preliminary steps can be taken to help decrease time to find the correct structure of NN.

Main question that should be answered is how many layers are needed and how many neurons should be used in each layer. A compromise between accuracy and generalization ability of NN must be made. Number of neurons should be enough to solve the task, but on the other hand it should not be too big to guarantee good generalization ability.

First decision, which should be made, is what kind of structure of NN (constricted or dilative) will be applied. The constricted NN has more neurons in the hidden layer than in the output layer. The dilative NN has more neurons in the input layer than in the hidden layer. The constricted NN is used to extract some features from the data set. The dilative NN well separates similar input vectors, but its generalization abilities are not high.

To find upper limit of number of neurons on hidden layer Kolmogorov theorem can be used [Hecht-Nielsen 1987]. According to this theorem any function with *n* variables can be approximated as superposition $2n+1$ one dimensional functions. The upper limit *h* can be found using formula (15). Parameter i is number of input elements.

$$h \leq 2i+1 \tag{15}$$

In other words there is no sense to use more hidden elements than doubled number of input elements. But this approach cannot be easily applied, because it requires proper selection of activation function (which is hand task).

[Eryurek & Upadhyaya 1990] used formula (16) to find absolute maximum of number of weights *w*.

$$w = i \log_2 p \pm i \tag{16}$$

In formula (16) parameter i is size of input vector and parameter *p* is number of vectors. For tasks that require good generalization, parameter *w* should be significantly less than that obtained from formula (16). Widrow and Lehr [Widrow & Lehr 1990] showed more accurate evaluation of number of weights for two-layer network:

$$\frac{N_w}{N_y} - K_1 \le C_d \le \frac{N_w}{N_y} \log_2 \frac{N_w}{N_y} + K_2 \tag{17}$$

In formula (17) parameter N_w is number of weights, parameter N_y is size of the output vector, parameter C_d is pattern capacity, parameters K_1 and K_2 are small positive numbers.

3.2 Self Organizing Maps

Self Organizing Map (SOM) [Kohonen 2001] is neural network model that maps high dimensional data onto small dimensional (usually two-dimensional) space. After using this algorithm similar vectors from the input space are located near each other in the output space. Mainly SOM is used for visual representation of input space and data preprocessing. SOM can be used to determine differences in behavior of investigated system. It might help to find abnormal modes in behavior of system. Also analysis of data with help of SOM can be useful for semantic interpretation of clusters. Mostly SOM is known as an unsupervised algorithm, but there is also a supervised implementation of SOM. The SOM combines itself both vector quantization [Kohonen 2001] and vector projection [Kohonen 2001] algorithms.

Usually SOM represents itself either hexagonal or rectangular grid of neurons. Each neuron is characterized by two vectors. The first vector represents coordinates in the input space and the second vector represents coordinates on the map. Each neuron is connected

to adjacent neurons by neighborhood function. The size of neighborhood is slightly decreased during training process. The hexagonal and the rectangular grids with different size of neighborhood are illustrated in figure 3.5.

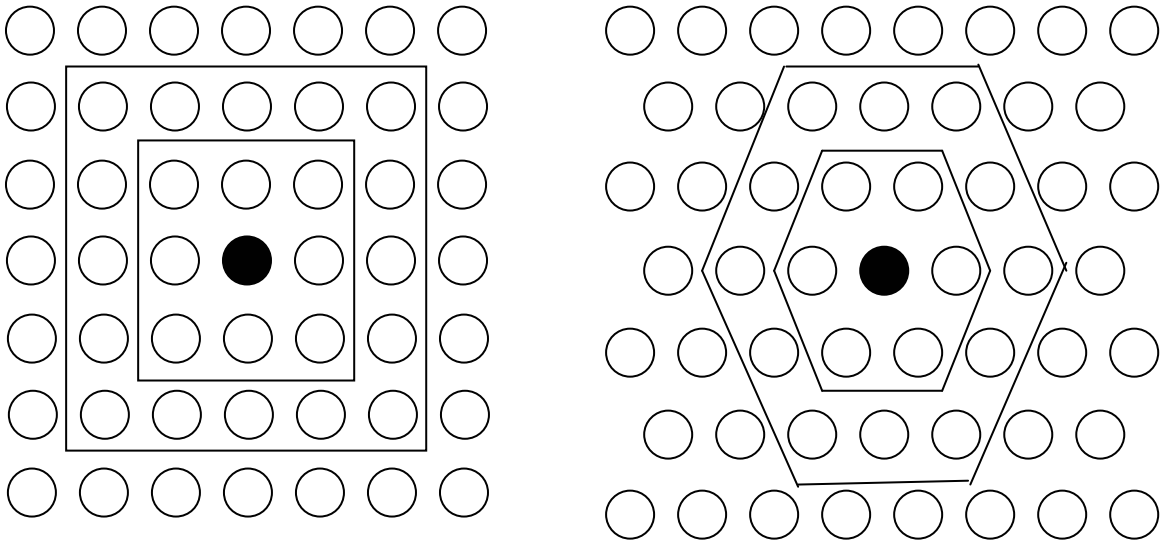

Figure 3.5 – The rectangular and the hexagonal structures with different neighborhood size

Topology of SOM is fixed from the beginning. Tutorial SOM toolbox for Matlab [Vesanto et al. 2000] suggests that the map should not have square form. One side of the map should be bigger than other approximately twice. Also it suggests using heuristics to define initial number of neurons:

$$Number \quad of \quad neurons = 5\sqrt{Number \quad of \quad samples} \tag{18}$$

Naturally, before starting the training algorithm, weights of neurons should be initialized. There are several means to initialize them:

- Random initialization
- Sample initialization
- Linear initialization

In the random initialization procedure weights are initialized with small random values. In the sample initialization procedure weights are initialized with random samples from data set. In the linear initialization procedure weights are initialized according to linear subspace spanned by the two principal eigenvectors from input data set.

Two approaches exist to train SOM. The first approach is sequential training. In this approach in each training iteration one sample is chosen randomly. After that winner-neuron *c* (also called as Best Matching Unit (BMU)) can be determined using formula (19).

$$\|x - m_c\| = \min_i \|x - m_i\| \tag{19}$$

In formula (19) one can understand we need to select the closest neuron m_c from all neurons m_i to the input vector *x*. After finding BMU the weight vectors should be updated. In this step not only weight of BMU should be updated, but also weights of neighbors of BMU should be updated. The rule for updating weight of some neuron i is defined by the following formula:

$$m_i(t+1) = m_i(t) + h_{ci}(t)(x(t) - m_i(t)) \tag{20}$$

In formula (20) $x(t)$ is randomly selected vector from data set at time t, $h_{ci}(t)$ is the neighborhood kernel. The neighborhood kernel is function that defines influence region of each sample on SOM. This function consists of two parts. The first part is neighborhood function $h(t)$, which depends on time and distance between map units. The second part is learning rate function $\alpha(t)$. There are quite a lot of variants of neighborhood functions. The most used is bubble and Gaussian neighborhood functions. Bubble function defines constant region for whole neighborhood. Gaussian neighborhood function is defined by formula (21).

$$h(t) = \exp\left(-\frac{\|r_c - r_i\|^2}{2\delta^2(t)}\right) \tag{21}$$

Where r_c and r_i are locations of BMU on the map and some neighboring neuron respectively.

Function $\alpha(t)$ can be determined using formula (22). Constants A and B are selected in such a way to guarantee good convergence of training algorithm.

$$\alpha(t) = \frac{A}{B+t} \tag{22}$$

The neighborhood kernel h_{ci} can be determined from formula (23).

$$h_{ci}(t) = h(t)\alpha(t) \tag{23}$$

Usually training process is divided into two parts. In the first phase, quite large neighborhood radius and large learning rate are used. In this phase neurons should roughly spread on the map according to the data. In the second phase, small neighborhood radius and small learning rate are used. In this phase neurons shift more precisely to take their final positions on the map. Example of one training step is illustrated in figure 3.6.

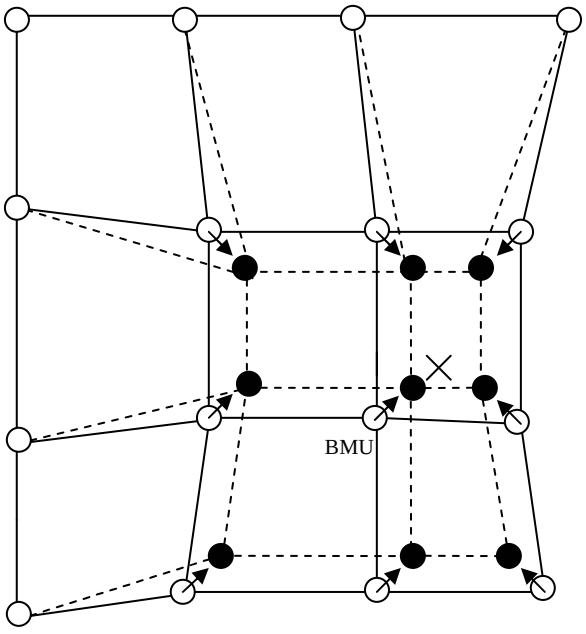

Figure 3.6 – Training example

In figure 4.6 circles denote neurons. Black circles denote neurons, which moved closer to input example denoted by X. These neurons were selected with help of neighborhood function. Thus not only weights of BMU are changed, but weights of the closest neurons are updated as well.

Another method that is used to train SOM is Batch algorithm [Kohonen 1993]. Batch algorithm was introduced by Teuvo Kohonen in 1993. Batch algorithm is parameter-free and very fast version of SOM. This algorithm is also iterative, but in contrast to sequential training, instead of one random sample, whole data set is used. During training data set is spread according to Voronoi regions. Voronoi region of some neuron i is the group of all vectors x to which it is the closest one. Voronoi region is defined by formula (24).

$$V_i = \{\, x \mid \|m_i - x\| < \|m_j - x\|,\ i \neq j \,\} \tag{24}$$

Formula (25) is used to calculate new weights of neurons.

$$m_i(t+1) = \frac{\sum_{j=1}^{n} h_{ic}(t) x_j}{\sum_{j=1}^{n} h_{ic}(t)} \tag{25}$$

In formula (25) $h_{ci}(t)$ is kernel function and x is input vector.

3.3 Matlab Environment

In this subsection Matlab [Matlab] environment is described. Basis functions of Neural network toolbox and SOM toolbox [SOM toolbox] are described as well. In this work we used SOM toolbox (developed in Helsinki University of Technology) instead of native Neural network toolbox, because native toolbox does not provide all the required functions for investigating and analyzing data.

Matlab is powerful tool for realization of different mathematical tasks, data processing and visualizing information. Matlab is widely used for decreasing time of development; it allows fast comparison of different results to find the best solutions.

MatLab provides the following features:

- Fast and efficient implementation of matrix calculus
- Graphics for representing information
- Interactive visual environment for programming
- Interfaces with external languages (C/C++, Java)
- Connection to databases

Special sets of functions (toolboxes) are present in Matlab. These toolboxes play main role for solving particular tasks. Matlab contains the following base toolboxes:

- Communication toolbox
- Control system toolbox
- Fuzzy logic toolbox
- Image processing toolbox
- Neural network toolbox
- Optimization toolbox
- Signal processing toolbox
- Spline toolbox
- System identification toolbox

Matlab has quite good Neural network toolbox. It allows modeling a lot of architectures of neural networks such as MLP, Radial Basis Function Networks, Recurrent networks and many others. To model MLP Matlab has a lot of different functions and parameters. Thus it is impossible to describe all possibilities of Matlab in one section. To give an idea how to build and simulate MLP, let us look through simple example, which represents itself skeleton code of simple MLP:

```
net = newff(minmax(P), [l1,l2,l3], {'a_function1','a_function2','a_function3'}, 'tr_method');

net.trainParam.show = 50;

net.trainParam.epochs = 150;
```

net.trainParam.goal = 1e-3;

[net,tr]=train(net, P, T);

Y= sim(net, P);

In the above example new MLP object was created using *newff()* function. Parameter *P* is the name of training data set. Function *minmax* calculates matrix, which contains minimum and maximum values from training data set. MLP has 3 hidden layers, this was defined by the second parameter (*[l1, l2, l3]*). Where parameters *l1*, *l2* and *l3* define the number of neurons in the first, in the second and in the third layers respectively. The third parameter (*{'a_function1','a_function2','a_function3'}*) defines activation functions for each layer. Parameter *a_function* defines name of activation function. This parameter can have one of the following values:

- *logsig* (sigmoid activation function)
- *tansig* (hyperbolic tangent activation function)
- *purelin* (linear activation function)

Parameter *tr_method* defines training method. There are a lot of training methods that are realized in Matlab, for example such as *trainb* (batch algorithm), *trainrp* (resilient BP method), *traingd* (gradient descent algorithm) and many others.

Matlab allows monitoring of training process. It displays a figure with error function. Updating frequency of the figure is defined by *net.trainParam.show* parameter. Number of epochs is defined by *net.trainParam.epochs* parameter. Parameter *net.trainParam.goal* defines desired threshold to stop training. Function *train(net, P, T)* is used to train MLP. Parameter *T* is the name of matrix with desired outputs of MLP. Function *sim* is used to simulate MLP.

Matlab also contains implementation of SOM, but as was mentioned earlier this implementation is not good. SOM toolbox from HUT provides qualitative GUI and realization of all necessary functions. Like in case with MLP implementation, describing of

all possibilities of this powerful toolbox might be compared with size of a book. Therefore only main features are described.

Naturally, structure that contains all necessary information about data set is needed. Function *som_data_struct* is used for this purpose. To train SOM different methods can be used such as *som_make*, *som_randinit*, *som_batch_train*, *som_seqtrain* and many others. SOM toolbox contains implementation of different clustering algorithms for example such as k-means and hierarchical clustering algorithm. Set of different auxiliary functions allows performing such actions like calculating BMU, distributing labels, modifying of data set and many others. SOM toolbox provides a lot of possibilities to visualize data. It allows visualizing different kinds of maps in different ways. Examples of some visualization of SOM can be found in chapter number five.

3.4 Discussion

MLP is well-known approximator of any function thus it can be used to build core of NeuroSearch algorithm. Determining of the correct structure of MLP is quite complex task, but as was shown in section 3.1.3 some preliminary steps can be taken. Since we want algorithm with good generalization properties some ways to avoid the overlearning problem are described in section 3.1.2. SOM is widely used tool for analyzing and preprocessing data. Therefore its using will be beneficial in our work. Matlab environment provides all required tools to do research part of work with high quality.

4 Evolutionary Computing and NeuroSearch Algorithm

In this chapter we describe strategies that were undertaken to build efficient algorithm for solving the resource discovery problem. This algorithm was developed as part of Cheese Factory project [Cheese]. The core of this algorithm is MLP, which processes different inputs to produce solution about further forwarding of query. Training of MLP for NeuroSearch algorithm is quite complex task, because initially we have only the input information and we know nothing about relationships between input components. Evolutionary Computing (EC) approach is used to solve this problem Therefore this chapter reviews methods of EC as well. Since we use EC we don't know anything about the decision mechanism of NeuroSearch. Thus we have to review existing NeuroSearch's problems, which are related to the decision mechanism. More detailed attention to the solving of these problems will be paid in chapter number five.

4.1 Introduction to Evolutionary Computing

Methods of EC or Genetic Algorithms (GA) were introduced by J. H. Holland [Holland 1975]. Nowadays GA is widely used to solve optimization tasks. Tuning of weights of NN is classical example of such tasks. Basis of GA is stochastic optimization. Therefore main disadvantage of these methods is that it is impossible to know beforehand how many iterations are needed to find an optimal solution.

Since GA has a lot in common with biology, we need brief introduction to some biological terms. Each biological individual is represented with its phenotype and genotype. In fact phenotype determines physical manifestation of an organism in real world. Genotype is internally coded inheritable information, which is carried by all living organisms. Each gene or in other words element of genotype's information has its representation in phenotype.

To solve some practical task we need to represent all properties of an object in form, which is suitable for GA. Mostly for this purpose simple bit strings are used.

Classical variant of GA uses two basis operators. The first operator is crossover. To perform crossover two individuals from population are selected. After that their chromosomes are divided at random point into two parts. The first part of the first chromosome connects to the second part of the second chromosome and the second part of first chromosome connects to the first part of second chromosome. Hereby we get two new individuals. This process is illustrated in figure 4.1.

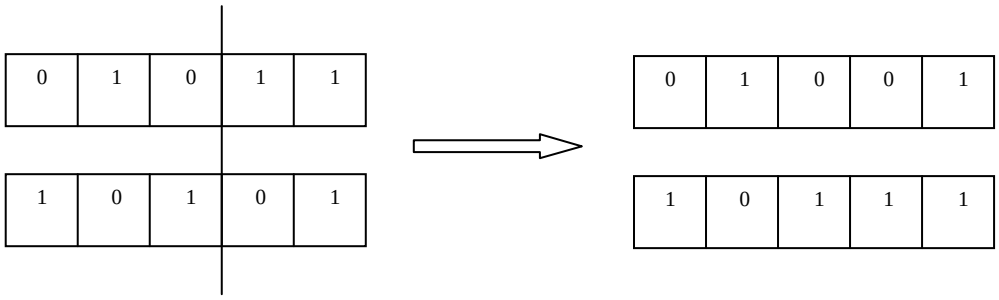

Figure 4.1 – Crossover

The second operator is mutation. To perform mutation one random bit is selected. Then this bit is inverted. Mutation is used to guarantee diversity of population and introduces new material to the population.

GA uses the following steps:

1. Initial population is formed randomly.
2. Calculating values of fitness function for all individuals in the population. This function determines how well the population at some moment is.
3. Selecting individual from population.
4. Selecting another individual according to crossover probability and perform crossover for these two individuals.
5. Performing mutation for new individuals according to probability of mutation.

6. Inserting new individuals to the population and removing some individuals with bad value of fitness function.

7. Executing operations for all individuals starting from step number three.

8. If new value of fitness function satisfies us then GA stops, else starting a new generation by executing of all operations starting from step number two.

The main part of GA, which significantly influences on its performance, is selecting individuals for crossover in steps number three and four. Most common way is to use of roulette wheel method [Chen & Smith 1999]. The idea of this method is the following: the size of the sectors on the roulette wheel for every individual is assigned according to fitness function of this individual. The use of this method increases probability to save phenotype of 'better' individuals and at the same time saves chances for weak individuals to pass to new generation. Another quite usable approach is tournament method [Blickle & Thiele 1995]. Using of this method is concluded in the following: several individuals are selected and individual with the best value of fitness function is selected. Some realizations of GA use strategy of elitism [Yamauchi & Randall 1994]. This strategy guarantees saving the best individuals for the next iterations. Strategy of elitism guarantees fast convergence of GA, but GA with this strategy might find local minimum instead of global.

4.2 Description of NeuroSearch

In this subsection description of NeuroSearch algorithm is given. The description mostly based on information given in [Vapa et al. 2004].

4.2.1 Architecture of NeuroSearch

Decision mechanism of NeuroSearch is based on 3-layer perceptron. The first hidden layer contains 16 neurons, the second hidden layer 4 neurons and the output layer 1 neuron. Hyperbolic tangent is used as an activation function in the first and the second hidden layers. Threshold activation function is used in the output layer. The queries spread through network to all neighbors according to decision mechanism of Neurosearch. If

targeted resource is found on some node then this node sends back reply using the same path with the query, which found this resource.

Figure 4.2 illustrates processing of NeuroSearch resource query.

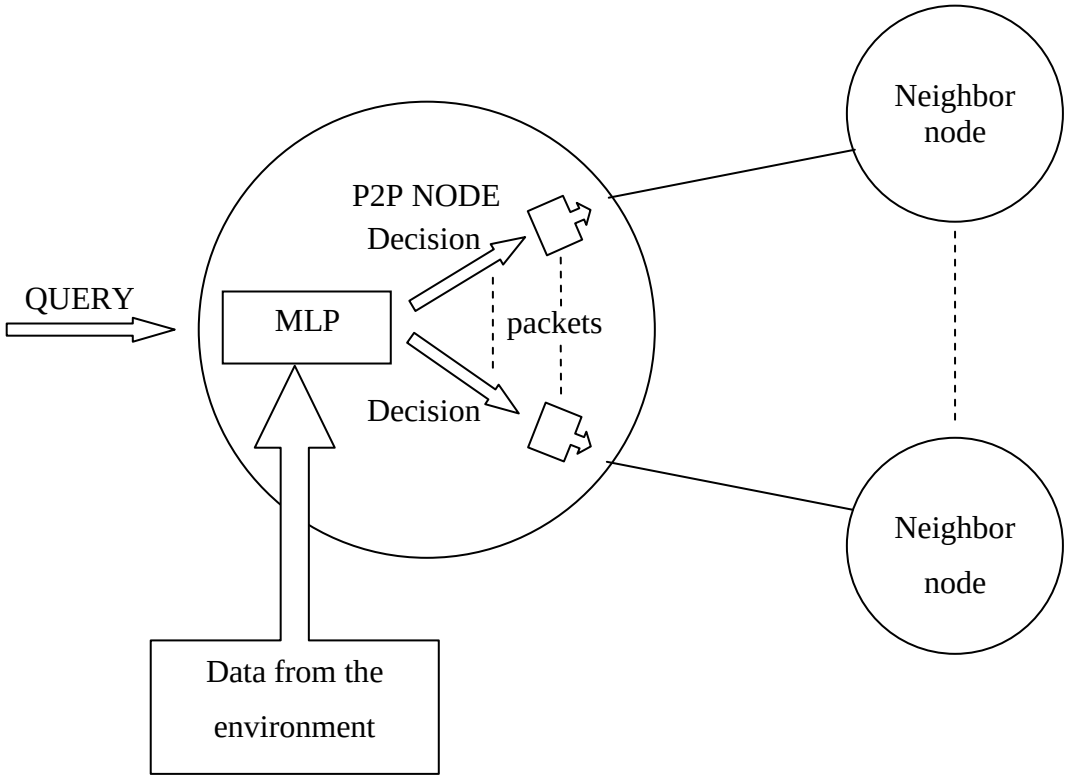

Figure 4.2 – Processing of NeuroSearch resource query

NeuroSearch algorithm is under active development, therefore structure of MLP and numbers of inputs are changed all the time. The original version of NeuroSearch included the following inputs:

- *Bias* = 1 is the bias term.
- *Hops* is the number of hops in the message.
- *NeighborsOrder* tells in which neighbor rank index this receiver is compared to other receivers. The receiver with the best rank has value of 0.
- *Neighbors* is the number of the receiver's neighbors.
- *MyNeighbors* is the number of node's neighbors.

- *Sent* has value 1 if the message has already been forwarded to the receiver using this link. Otherwise it has value of 0.
- *Received* has value 1 if the message has been received earlier, else it has value of 0.

Naturally, all inputs should be normalized, in other words they should belong to interval from 0 to 1. To understand why normalization is needed, let us consider an example. Assume that we have MLP with two inputs. The first input processes values, which belong to interval from 0 to 2. The second input processes values, which belong to interval from 100 to 1000. Using of non-scaled input information lead us to situation where the first input will be totally useless, because only values from the second input will satiate the output with themselves. Normalization function for inputs *Hops* and *NeighborsOrder* is defined by formula (26).

$$f(x) = \frac{1}{x+1} \tag{26}$$

Inputs *Neighbors* and *MyNeighbors* are scaled with function, which is defined by formula (27).

$$f(x) = 1 - \frac{1}{x} \tag{27}$$

4.2.2 Training of NeuroSearch

The topology of the P2P network represents itself power-law graph, which was generated using Barabasi-Albert model [Barabasi & Albert 1999] [Barabasi 2002]. The graph obeys power-law networks' neighbor distribution (2). In formula (2) for Barabasi – Albert model γ parameter is equal to 3. The graph contains 100 nodes. The highest degree node has 25 neighbors.

Decision mechanism of NeuroSearch is based on MLP. Since we do not know exact relationships between input parameters and correct value of target function we cannot use usual strategies to train MLP. As will be mentioned in section 5.1, GA is widely used

method to adjust weights of NN. The main advantage of this method is that we can get well trained NN even if we do not know anything about how inputs affect on certain decision. There is only one thing that we need and it is fitness function. NeuroSearch uses the following general fitness function in its training process:

$$fitness_h = \sum_{j=1}^{n} score_j \tag{28}$$

Where h denotes index of neural network and j denotes index of some query.

The following rules are used to calculate *score*:

1. If $found\ resources \geq \frac{available\ resources}{2}$ then $score = 50 * \frac{available\ resources}{2} - packets$: when half of the available resource instances are found from the network, the fitness value does not grow if neural network locates more resources.

2. If $found\ resources < \frac{available\ resources}{2}$ and $found\ resources > 0$ then $score = 50 * found\ resorces - packets$: if the number of found resources is not enough then the neural network develops if it locates more resources.

3. If $found\ resorces = 0$ then $score = 1 - \frac{1}{packets + 1}$: if none of the resources are found then the neural network should increase the number of packets sent to the network.

4. If $packets > 300$ then $score = 0$: an algorithm that eventually stops is always better than algorithm that does not.

Where parameter $available\ resources$ denotes number of resource instances that can be found from the network. Parameter $found\ resorces$ denotes amount of replies from nodes

where targeted resource was found by the neural network. Parameter *packets* denotes total amount of packets, which were used to locate targeted resources.

To train NeuroSearch only mutation operator is used. This is done by random variation using normal distribution. The random variation function is defined with the following formulas:

$$\sigma_i^{'}(j) = \sigma_i(j)\exp(\tau N_j(0,1)),\ j = 1,...,N_w \tag{29}$$

$$w_i^{'}(j) = w_i(j) + \sigma_i^{'}(j)N_j(0,1),\ j = 1,...,N_w \tag{30}$$

$$\tau = \frac{1}{\sqrt{2\sqrt{N_w}}} \tag{31}$$

Where N_w is the total number of weights, $N_j(0,1)$ is a standard Gaussian random variable resampled for every j, σ is the self-adaptive parameter vector for defining the step size for finding the new weight and $w_i^{'}(j)$ is the new weight value.

4.2.3 Simulation Results

Simulation of NeuroSearch showed that it is able to locate approximately half of resources from the P2P network. Mostly NeuroSearch locates more resources than BFS algorithm with TTL 2 and in the same time less than BFS algorithm with TTL 3. Behavior of NeuroSearch is quite similar to BFS with TTL 2 in the core of the P2P network, but better in the edges. Since the goal of NeuroSearch is to locate only half of available resources, therefore BFS algorithm with TTL 3 locates significantly more resources than NeuroSearch. But BFS with TTL 3 uses much more packets to locate these resources. Efficiency of BFS algorithm significantly depends on connectivity of node, which starts the query. But simulation shows that Neurosearch is independent on connectivity of starting node.

4.3 Discussion

Despite of time consumption, applying of GA to train NeuroSearch is quite useful approach and this is probably the only possible way to group all components (input information) to make decisions about further forwarding without knowledge of correct relationships between components in the power-law networks. The main problem of NeuroSearch algorithm is that we do not know anything about its decision mechanism. This causes unclear situation with problem definition and quite weak basis for chosen neural network architecture. These problems were avoided using EC in training process, but not actually solved. Therefore it is quite difficult to say about theoretical performance maximum of NeuroSearch algorithm. Simulation results show that NeuroSearch algorithm has quite good performance. But it is roughly seen from simulation of NeuroSearch that it has some problematic areas (for instance core part of the P2P network) where it makes sometimes non-optimal decisions. These areas should be investigated more carefully. More detailed attention to all these questions will be paid in chapter number five.

5 Data Analysis

In this chapter data analysis techniques are proposed to investigate behavior of NeuroSearch algorithm in the space of its decisions. The SOM [Kohonen 2001] is used to analyze data obtained from simulation of NeuroSearch. Data represents itself results of simulation of 100 queries started from each node. Parameters of the P2P network are the same that were described in section 4.2.2. Data is stored in XML files. In these XML files each decision is represented by 16 input parameters (including bias), which were used as an input vector for the MLP. Also in these XML files all these 16 input parameters are put in correspondence to output of the MLP. To extract information from these files XML parser was written using PHP language. To analyze the extracted data SOM toolbox [SOM toolbox] for Matlab [Matlab] was used.

5.1 Objectives of Data Analysis

Since we do not know the decision mechanism of NeuroSearch we cannot say anything about the complexity of resource discovery in P2P system. Naturally, each component (e.g., query hops, number of neighbors) in the system has its own properties, which might depend on the state of the system (in other words on other components). If behavior of the system is sum of all components then the system is simple, even if the system has a lot of components. Such system can be described with a set of equations. Some systems have unclear dependencies and their behavior and properties cannot be analyzed separately or removing of components causes loss of principal properties. In this kind of cases we are dealing with complex system. The main question that arises here is what kind of system resource discovery problem in peer-to-peer network is. If resource discovery can be classified as simple system it would be beneficial to use analytical model that contains rules and equations to describe the behavior of the system.

Developed algorithm (NeuroSearch) can be considered as the main part of information model of the system, which utilizes different properties of P2P environment to find efficient strategy for resource discovery problem. To build this information model black box method was used. The method was introduced by Wiener [Wiener 1948]. In contrast to

analytical approach, where inner structure is modeled in the black box method external behavior of the system is modeled. Neural networking combined with evolutionary computing is quite common method for building black boxes. Naturally, using our information model (neural network inputs) it is quite difficult to explain its behavior.

To answer the question about complexity of the resource discovery problem we need to solve inverse task. In other words is it possible to explain behavior of the system using analysis of input-output pairs of the system? Also to evaluate the quality of algorithm we need to evaluate the correctness of inverse task because algorithm without general abilities becomes useless. Correct task requires that the following conditions are met:

- Existence of solution for whole data set
- Existence of unique solution
- Stability of exact solution to little changes of data set

The SOM is widely used approach for data mining and feature selection. Good illustration of using SOM for data analysis is given in [Kaski et al. 1999] and [Vesanto & Alhoniemi 2000]. Since the SOM combines vector quantification algorithm and projection algorithms, the SOM suits quite well for our analysis. One of the SOM properties is that near vectors from the input space will be positioned near each other in the output space. This helps to investigate existing relations between input components and their influence on particular decision of NeuroSearch. Also we can investigate clustering structures and find possible decision boundaries, which describe behavior of NeuroSearch.

5.2 Investigation of NeuroSearch

In contrast to the version of NeuroSearch that was described in [Vapa et al. 2004], in this chapter we investigate new version. The difference between two versions is the number of the MLP inputs. To provide more efficient search in the P2P network the following inputs were added:

- *From* has value 1 if the current message was received from this receiver. Otherwise it has value of 0.

- *toUnsearchedNeighbors* tells how many unsearched neighbors the receiver is expected to have. If the query has not traveled through the receiver earlier then the value is equal to *Neighbors* minus one. Otherwise the query has registered how many neighbors were unsearched when visiting the receiver. If the current node was not visited earlier then this input is equal to registered number of unsearched neighbors minus one. If the current node was visited earlier then this input is equal to registered number of unsearched neighbors.
- *initiatorNeighborAmount* is the number of the query initiator's neighbors.
- *fromNeighborAmount* is the number of the previous sender's neighbors.
- *repliesNow* is the number of replies the query has located in its query path.
- *packetsNow* is the number of packets the query has produced in its query path.
- *repliesToGet* is the number of resources that needs to be located.
- *toVisited* has value 1 if the message has already traveled through the receiver earlier. Otherwise it has value of 0.
- *randomNeighbor* has value 1 for randomly selected receiver and 0 for other receivers in the current node.

Normalization function for inputs *toUnsearchedNeighbors*, *repliesNow* and *packetsNow* is defined by formula (32). Normalization function for inputs *fromNeighborAmount*, *initiatorNeighborAmount* and *repliesToGet* is defined by formula (27).

$$f(x) = 1 - \frac{1}{x+1} \tag{32}$$

To investigate clustering structure of decision mechanism of NeuroSearch U-matrix was built. U-matrix is commonly used to illustrate the output map. U-matrix shows distance between neighboring units on the map. Therefore we can use it to investigate clustering structure of the output map and data as well. To do this we also put labels and 'hit' units on the output map. NeuroSearch is built in such way that its decisions are separated on two semantic groups:

- If output of MLP has negative value then NeuroSearch is not forwarding the query further.
- Otherwise it is forwarding.

To investigate behavior of NeuroSearch more accurately we divide each of these two groups into three parts:

- If $output\ of\ MLP \leq -3$ then decision belongs to class ‘0’
- If $-3 < output\ of\ MLP \leq -1$ then decision belongs to class ‘1’
- If $-1 < output\ of\ MLP < 0$ then decision belongs to class ‘2’
- If $0 \leq output\ of\ MLP < 1$ then decision belongs to class ‘3’
- If $1 \leq output\ of\ MLP < 2.5$ then decision belongs to class ‘4’
- If $output\ of\ MLP \geq 2.5$ then decision belongs to class ‘5’

One can see that classes 0, 1 and 2 are responsible to ‘not forwarding’ decisions and classes 3, 4 and 5 respond to ‘forwarding’ decisions. Not equal intervals were selected to investigate behavior of NeuroSearch more thoroughly especially in areas around zero, where NeuroSearch has boundary on decisions. The total interval of the output of the MLP is concluded approximately in range from -5 to 4. Therefore we selected different intervals for negative and positive decisions to guarantee equal distribution of decisions between homogeneous classes. The range of negative classes is equal to 2. The range of positive classes is equal to 1.5. To investigate more carefully decisions around boundary between positive and negative decisions we selected smaller interval for classes that are responsible to these decisions, this interval is equal to 1. Our assumption is based on that decision of NeuroSearch in numerical format is somehow correlated with probability to forward the query further.

‘Hit’ units show how many and which decisions are near each other, in other words which units make cluster structure. ‘Hit’ units and classes have the following color correspondence:

- Blue color (■) accords to class ‘0’

- Cyan color ( ) accords to class ‘1’
- Green color ( ) accords to class ‘2’
- Yellow color ( ) accords to class ‘3’
- Magenta color ( ) accords to class ‘4’
- Red color ( ) accords to class ‘5’

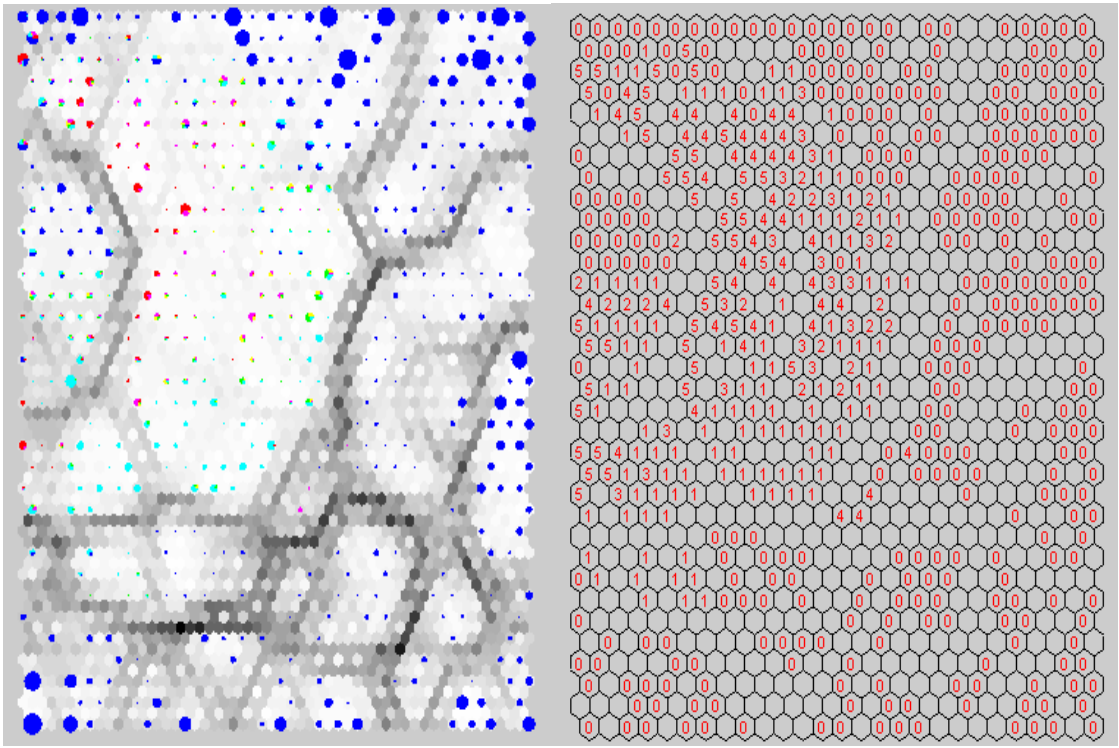

Figure 5.1 – U-matrix

Figure 5.1 illustrates U-matrix with ‘hit’ units and labels, which are pointed on different classes. Labels were distributed on the U-matrix using ‘vote’ system. According to this system the class of each neuron is selected based on the maximum amount of entries belonging to one class in the neuron’s region. Therefore the neuron gets the color that is most representative to the data elements that map into the neuron’s cluster.

In figure 5.1 one can see there are a lot of clusters on the output map. To know which variables are responsible for these clusters, we built component plane. This component plane shows what variable values of data points belong to particular cluster.

Figure 5.2 shows component plane. In the figure all 15 inputs excluding bias are shown. Analyzing U-matrix with distribution of 'hit' units together with component plane we are able to say that NeuroSearch has learned that it is not efficient to send the queries when at least one of variables *From, toVisited* and *Sent* equals to one. Also when *currentVisited* variable is equal to one NeuroSearch mostly does not send the queries further. But there are some samples (with *currentVisited* variable is equal to one) where NeuroSearch sends the queries further. It might be caused by low quality training, because optimization process is based on randomness had not many generations and good fitness value might have not been reached. Thus we are able to say that these four inputs (*From, toVisited, Sent* and *currentVisited*) are used to stop the queries.

From component plane we are able to see that *toUnsearchedNeighbors* and *Neighbors* variables are correlated. They have similar color distribution on the output map. Also we are able to see high correlation between *packetsNow* and *Hops* variables. This dependency can be easily explained. Number of packets grows when number of hops is increasing. Variables *fromNeighborAmount, packetsNow* and as consequence *Hops* are correlated somehow. This might be caused by the fact that NeuroSearch has tendency in general to forward queries to most connected nodes.

Analyzing *Neighbors* and *toUnsearchedNeighbors* variables it is possible to say that NeuroSearch does not forward the queries further if these inputs have small values. To study decision mechanism of NeuroSearch more thoroughly, four inputs (*From, toVisited, Sent* and *currentVisited*) were not taken into account (because we know about their influence on decision mechanism). Also our further investigation is based on *Hops* variable, because only this variable shows the state of algorithm in particular time interval, in other words analyzing intervals of this variable we are able to monitor the queries through their path. To do this we removed all samples where at least one of these four variables is equal to 1 and samples where *Hops* value is different from the one selected for the current investigation.

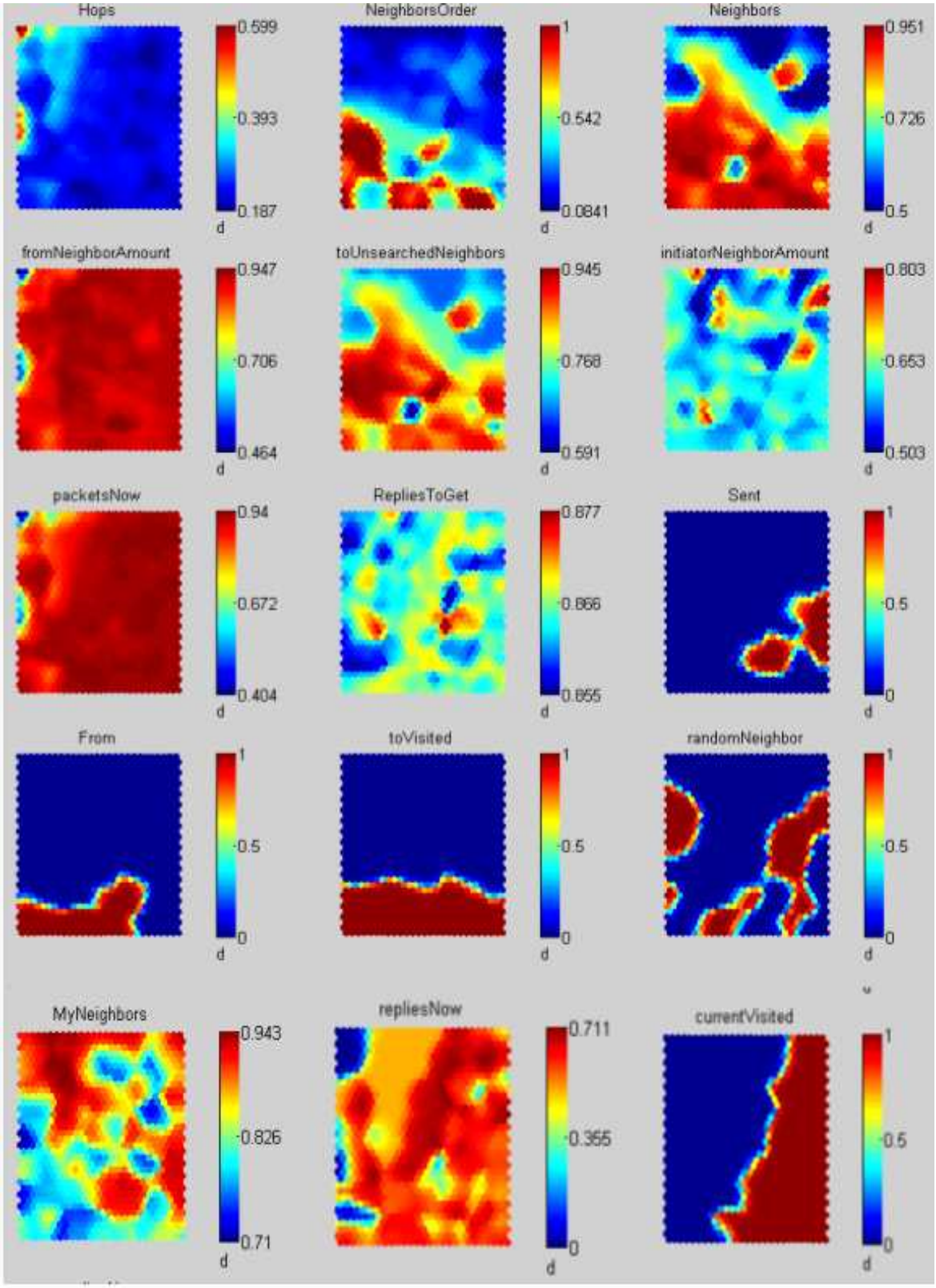

Figure 5.2 - Component plane

In the study of NeuroSearch was also noticed that NeuroSearch has 7 hops as maximum number of *Hops* in the message. Thus we separate our investigation on 7 parts. Each case is for its own value of *Hops.*

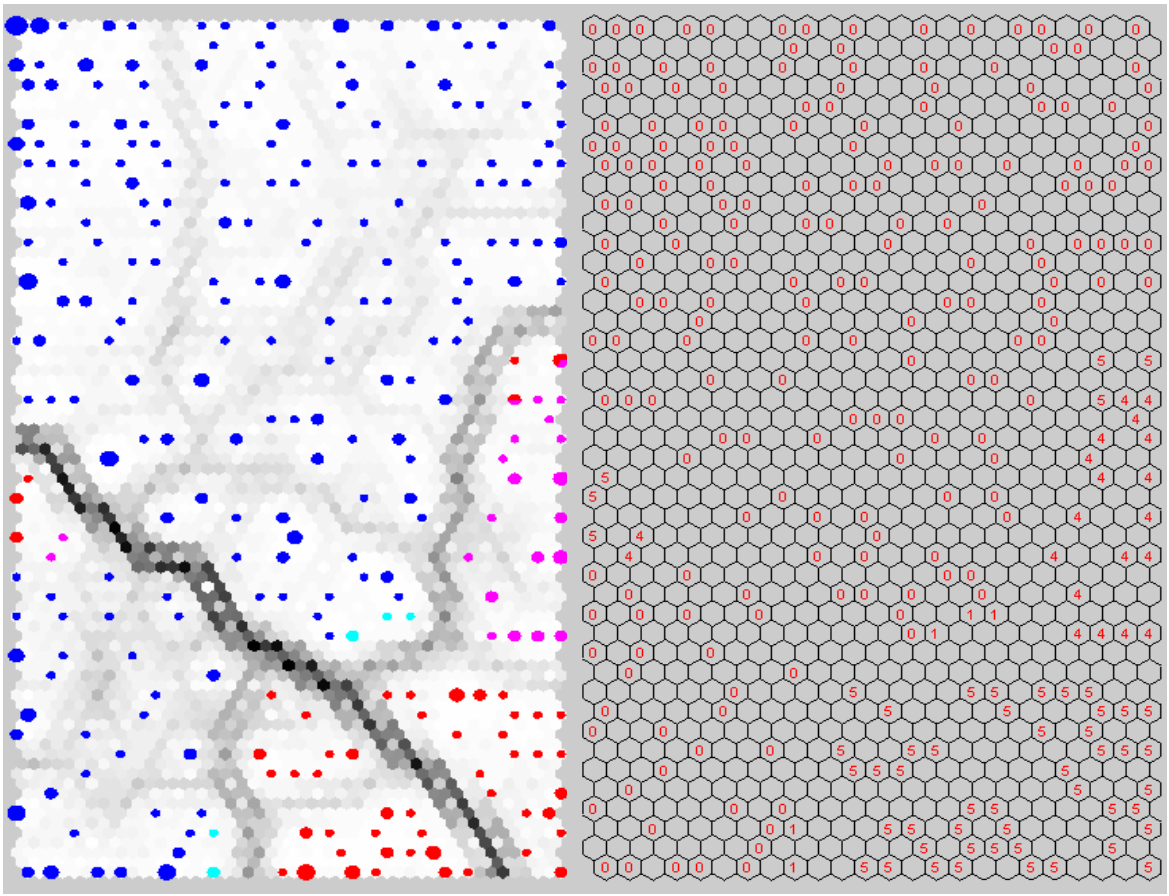

Figure 5.3 – U-matrix for *Hops* value is equal to 1

Analyzing U-matrix for *Hops* value is equal to 1 it can be seen that we have quite good partitioning of decisions. Mostly classes that correspond to decisions about forwarding are located in the bottom right cluster. Also couple of these classes are located on the left part. To know which variables are responsible for this partitioning we built component plane.

Figure 5.4 shows component plane for *Hops* value is equal to 1. Analyzing component plane it is easy to see that for right bottom cluster *NeighborsOrder* variable is responsible. In this cluster this variable has the highest value. Therefore only the highest ranking node will get the query at the first step. For forwarding decisions, which are stored on the left we

have more complex situation. For this region we are able to say that in this region we have some quite complex interaction between almost all variables. To extract this subcluster we are able to say that variable *randomNeighbor* should be equal to 1, variable *repliesToGet* should be quite high, variable *Neighbors* should have the smallest value and variable *MyNeighbor* should have high value.

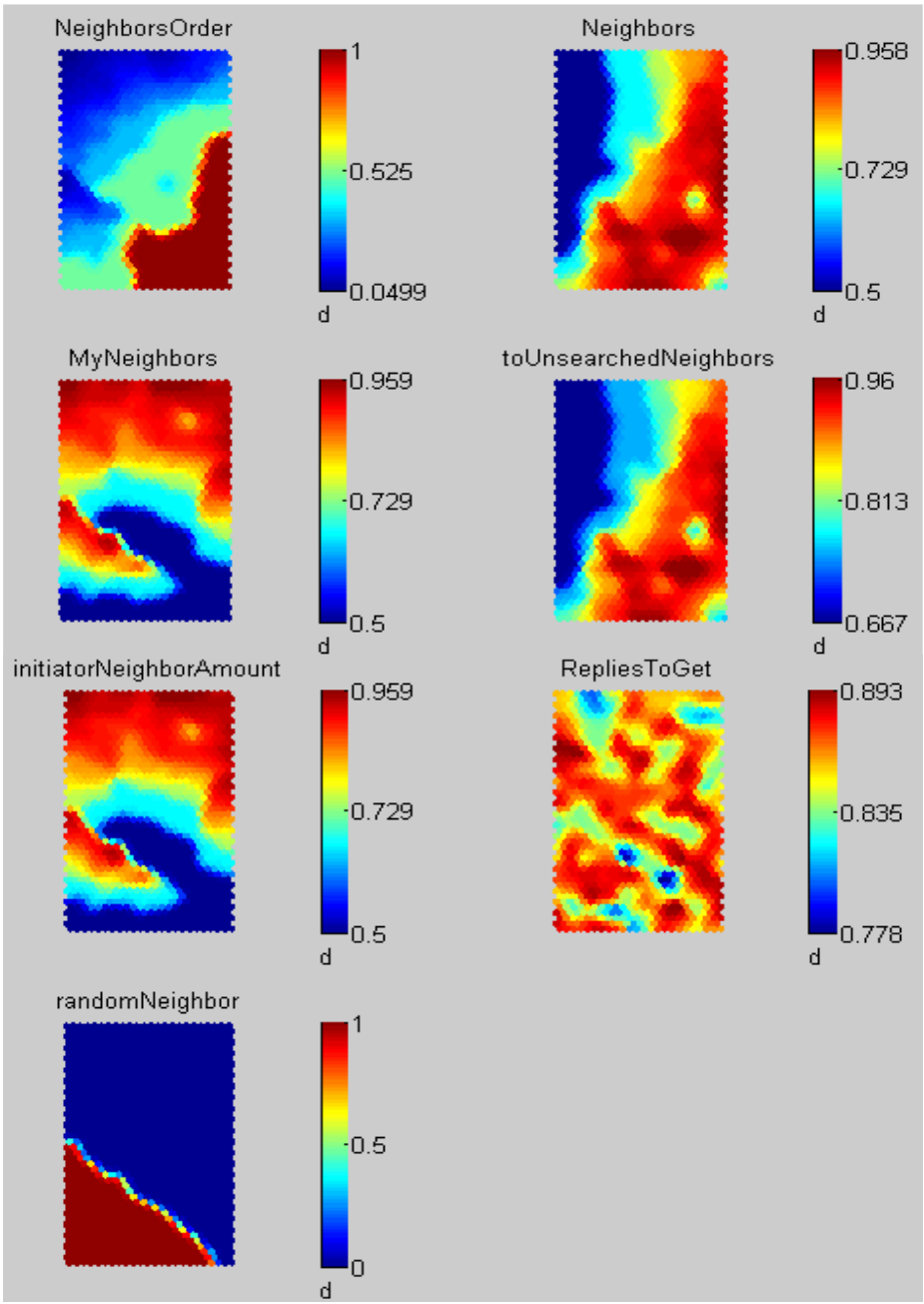

Figure 5.4 – Component plane for *Hops* value is equal to 1

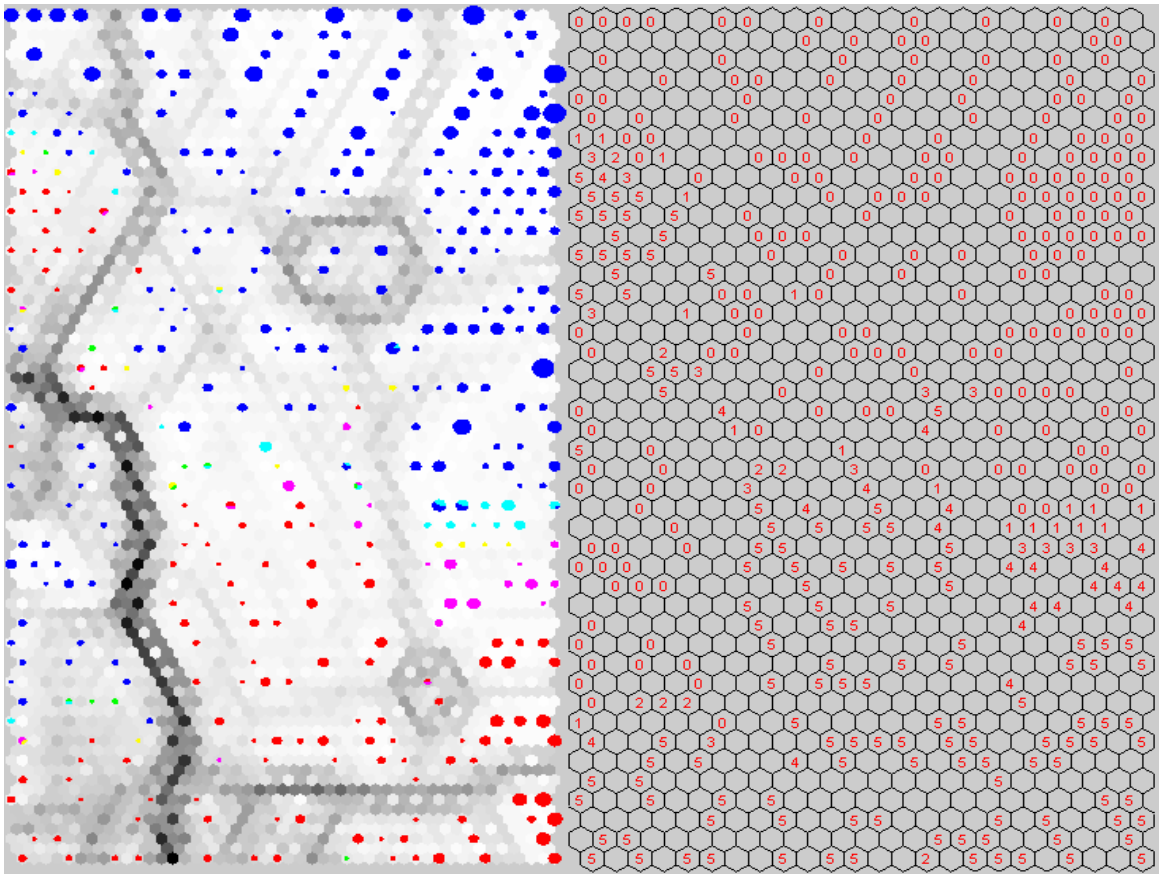

Figure 5.5 - U-matrix for number of hops in the message is equal to 2

Figure 5.5 shows U-matrix for the hops value in the message is equal to 2. From Figure 5.5 one can see that mostly classes that respond to decisions about forwarding are located in the bottom of U-matrix. Also we can see some classes that correspond to decisions about forwarding on the upper left side. To determine the influence of each variable to particular decision we built component plane for hops in the message is equal 2. Figure 6.6 shows this component plane. Analyzing this component plane one can see that decisions are based according to *fromNeighborAmount* and *Neighbors* or *initiatorNeighborAmount* variables. Naturally *fromNeighborAmount* and *initiatorNeighborAmount* variables represent themselves the same information because initiator and previous sender of the query is the same node. NeuroSearch prefers to send the queries to most connected neighbors. Also decisions of NeuroSearch depend a little on *fromNeighborAmount* or *initiatorNeighborAmount* variable. These variables 'move' decision boundary a bit. The

higher value of *fromNeighborAmount* variable is allowed to forward packets to little less connected nodes. Influence of other variables is minimal.

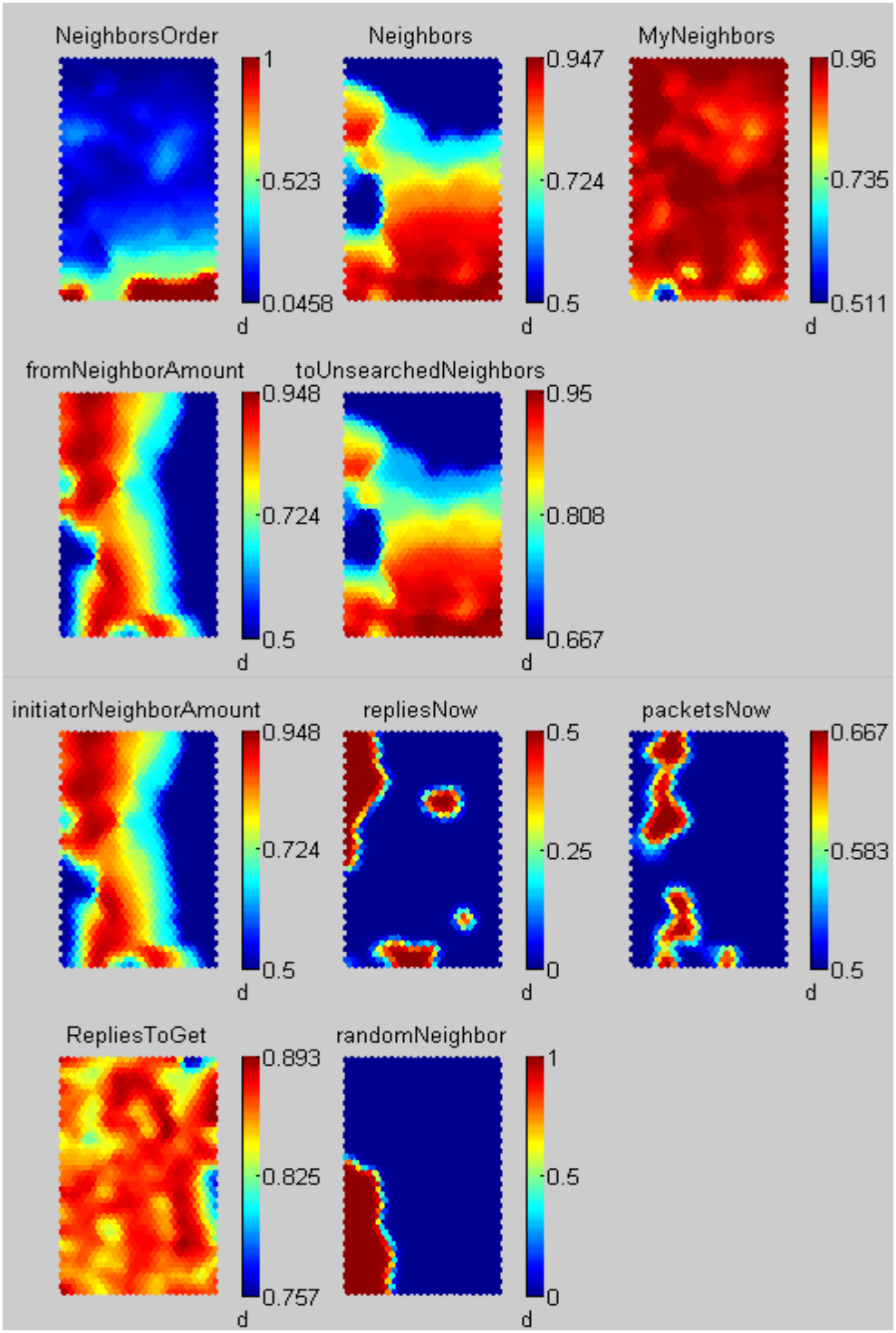

Figure 5.6 – Component plane for number of hops in the message is equal to 2

Figure 5.7 shows U-matrix for the hops value in the message is equal to 3.

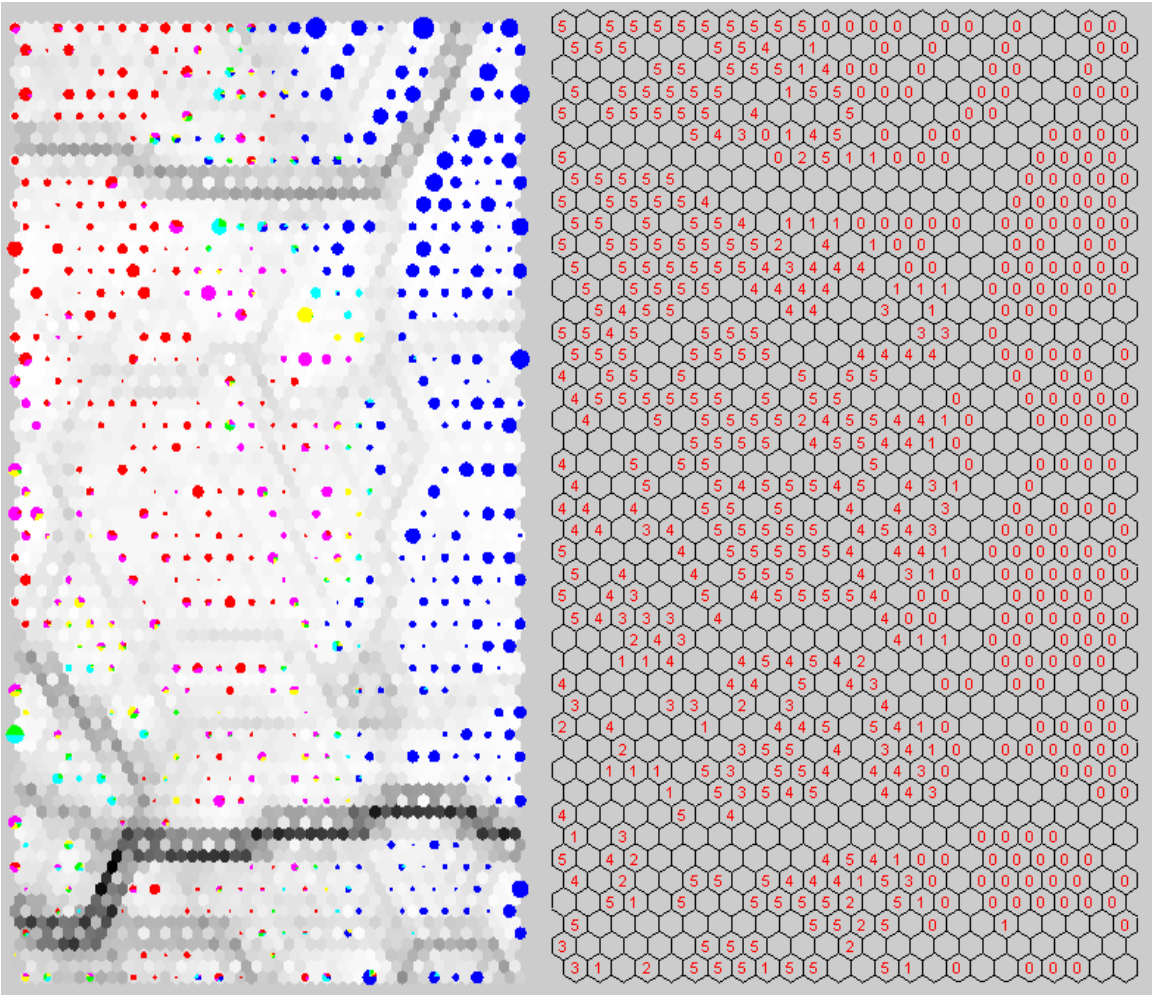

Figure 5.7 - U-matrix for number of hops in the message is equal to 3

Analyzing U-matrix one can see that classes that correspond to decisions about forwarding are located mostly on the left part of the U-matrix. Here we are not able to see strict separation between classes. Here we are only able to say about general tendencies. To know which variables correspond to particular decision we built component plane. Figure 5.8 illustrates component plane for number of hops in the message is equal to 3. Analyzing this component plane one can see that decisions as in the previous case are mostly based on *Neighbors* or *toUnsearchedNeighbors* variables. NeuroSearch does not forward the queries to low-connected nodes.

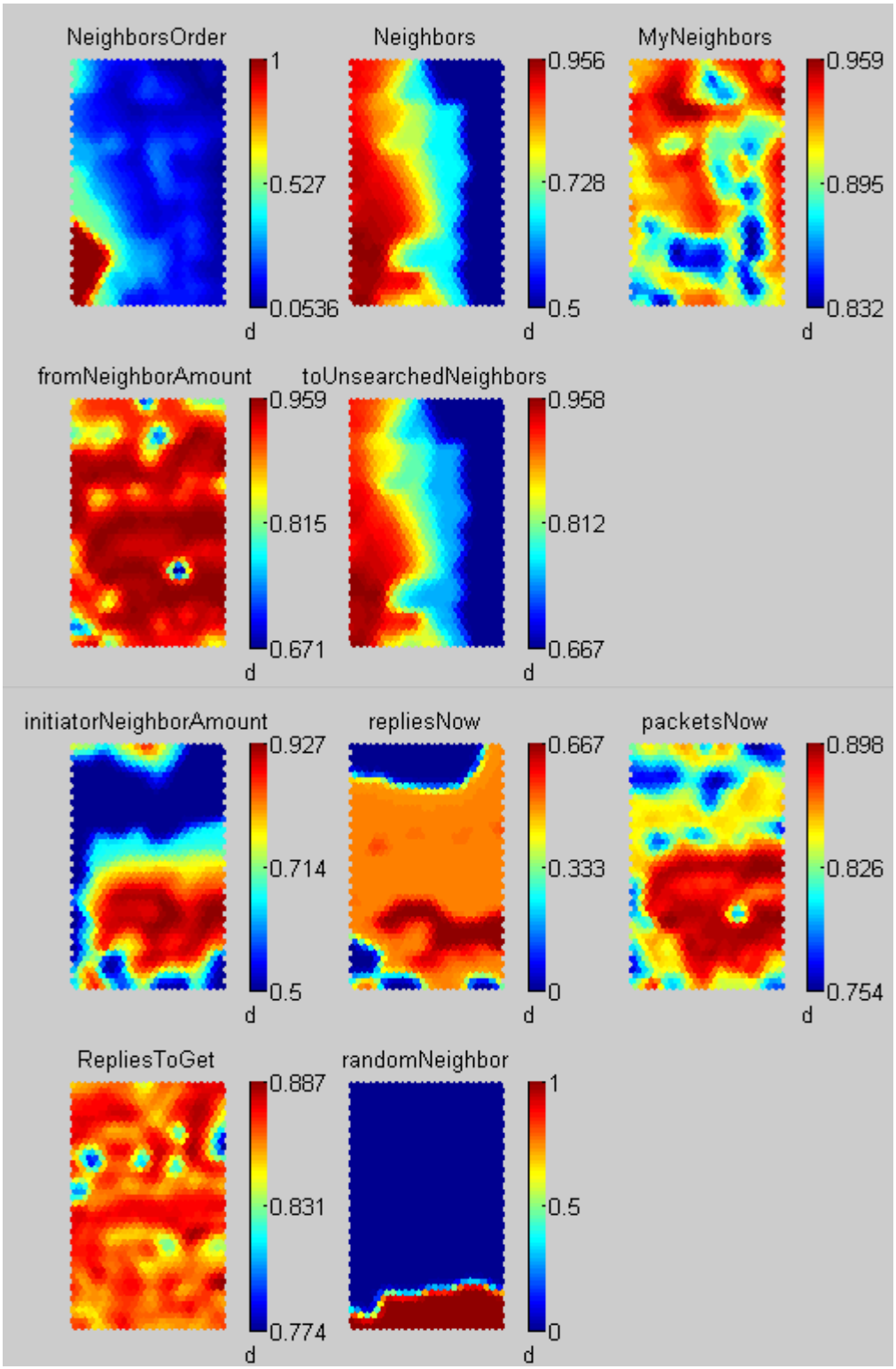

Figure 5.8 – Component plane for number of hops in the message is equal to 3

NeuroSearch has quite unclear behavior when variable *NeighborsOrder* is high. It is hard to say exactly what cause for particular decision in this area is. Influence of other variables is minimal.

Figure 5.9 illustrates U-matrix for the hops value in the message is equal to 4.

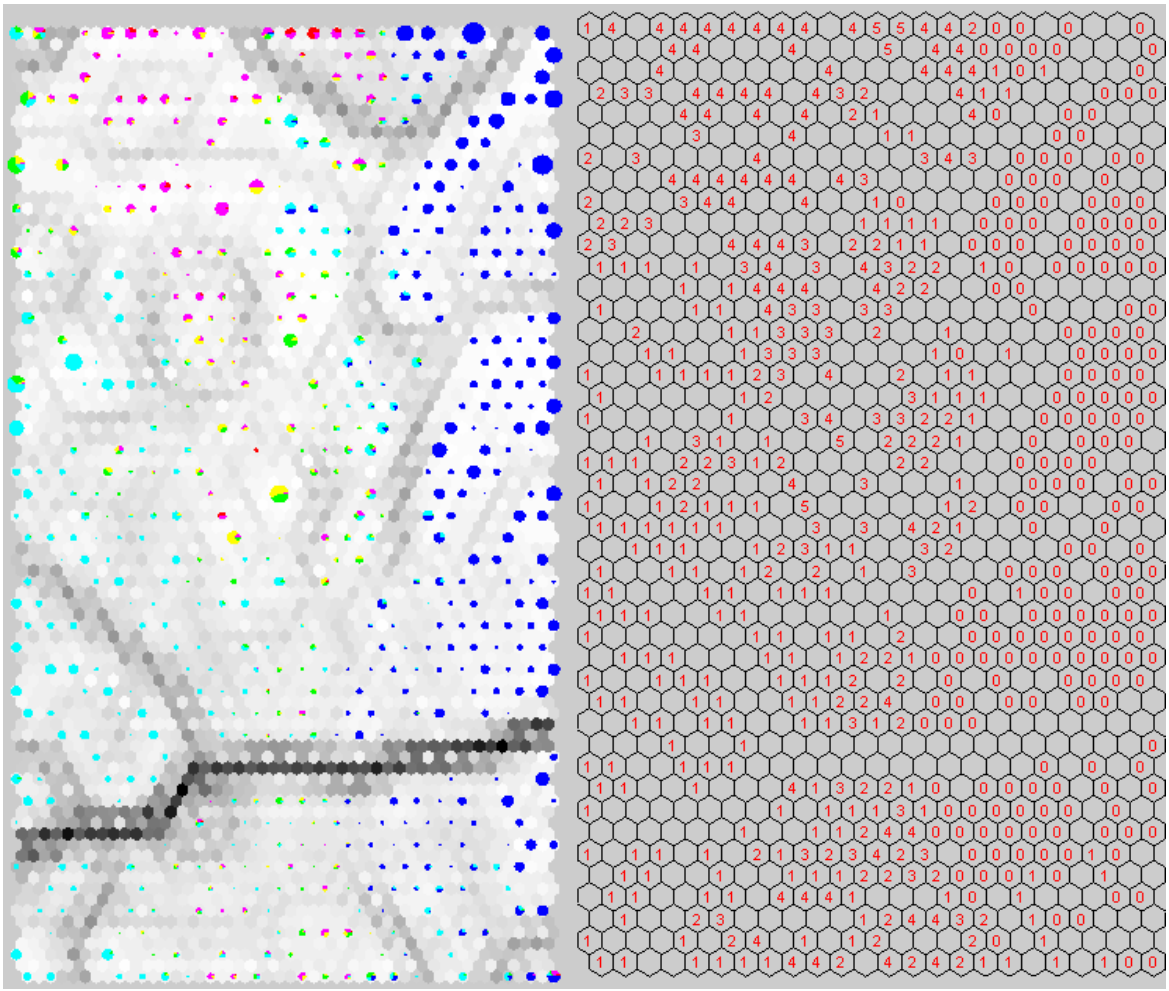

Figure 5.9 - U-matrix for number of hops in the message is equal to 4

Analyzing U-matrix one can see that there is no strict order in the data. Mostly classes, which correspond to positive decisions about forwarding, are located in the upper central zones. Some of these classes are overlapped. To know which variables are responsible for these decisions component plane were built. Component plane is shown in Figure 5.10.

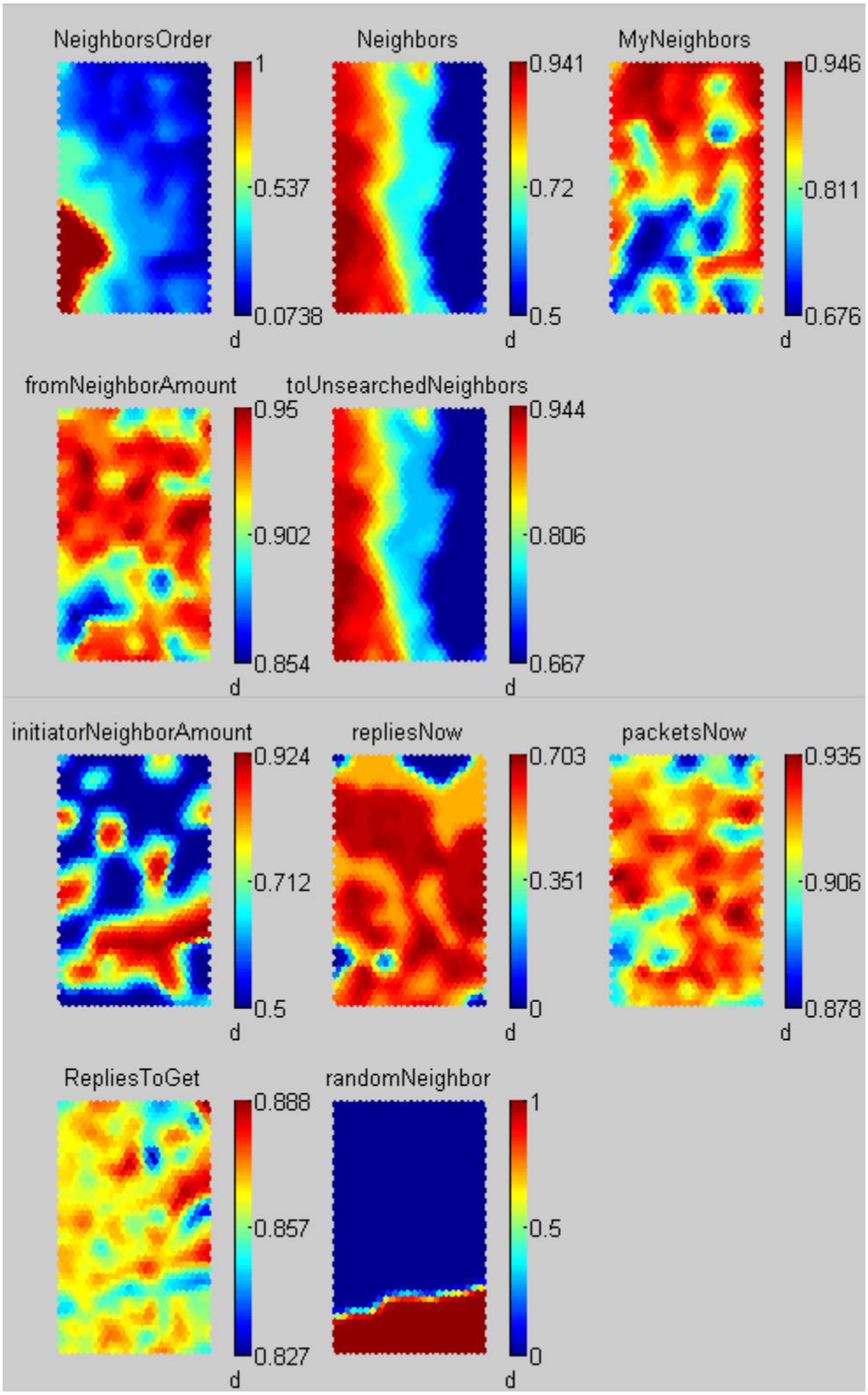

Figure 5.10 – Component plane for number of hops in the message is equal to 4

Analyzing this component plane one can see that decisions mostly are based on *Neighbors*, *toUnsearchedNeighbors* and *NeighborsOrder* variables. NeuroSearch does not forward the queries to low-connected nodes. Also NeuroSearch does not forward the queries to high connected nodes if *NeighborsOrder* variable is high. Influence of other variables is not so big. The area where *randomNeighbor* variable is equal to 1 is highly overlapped with different classes.

Figure 5.11 illustrates U-matrix for the hops value in the message is equal to 5.

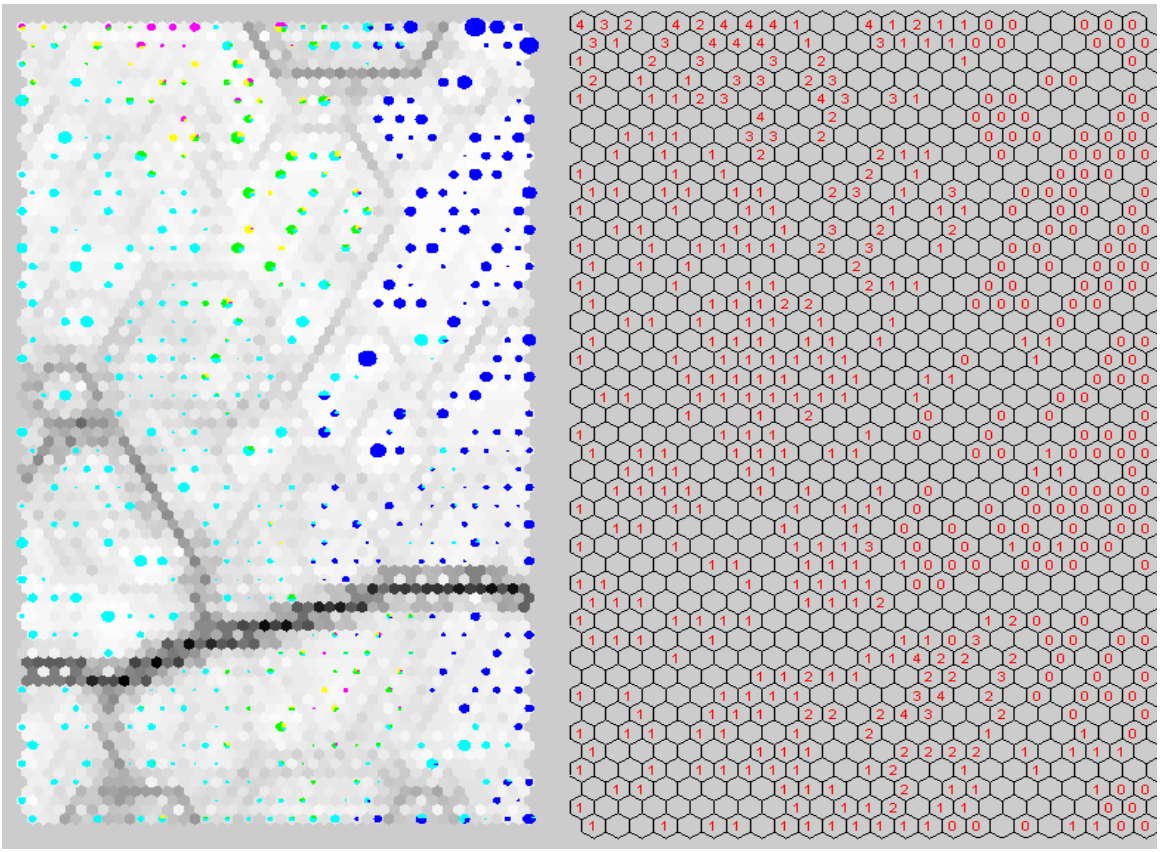

Figure 5.11 - U-matrix for number of hops in the message is equal to 5

Analyzing U-matrix one can see that there are not so much classes that correspond to positive decisions about forwarding. Some of the classes are overlapped. Especially those, which are related to decision boundaries (the output of the MLP is around zero).

Component plane for case where number of hops in the message is equal to 5 is shown in Figure 5.12.

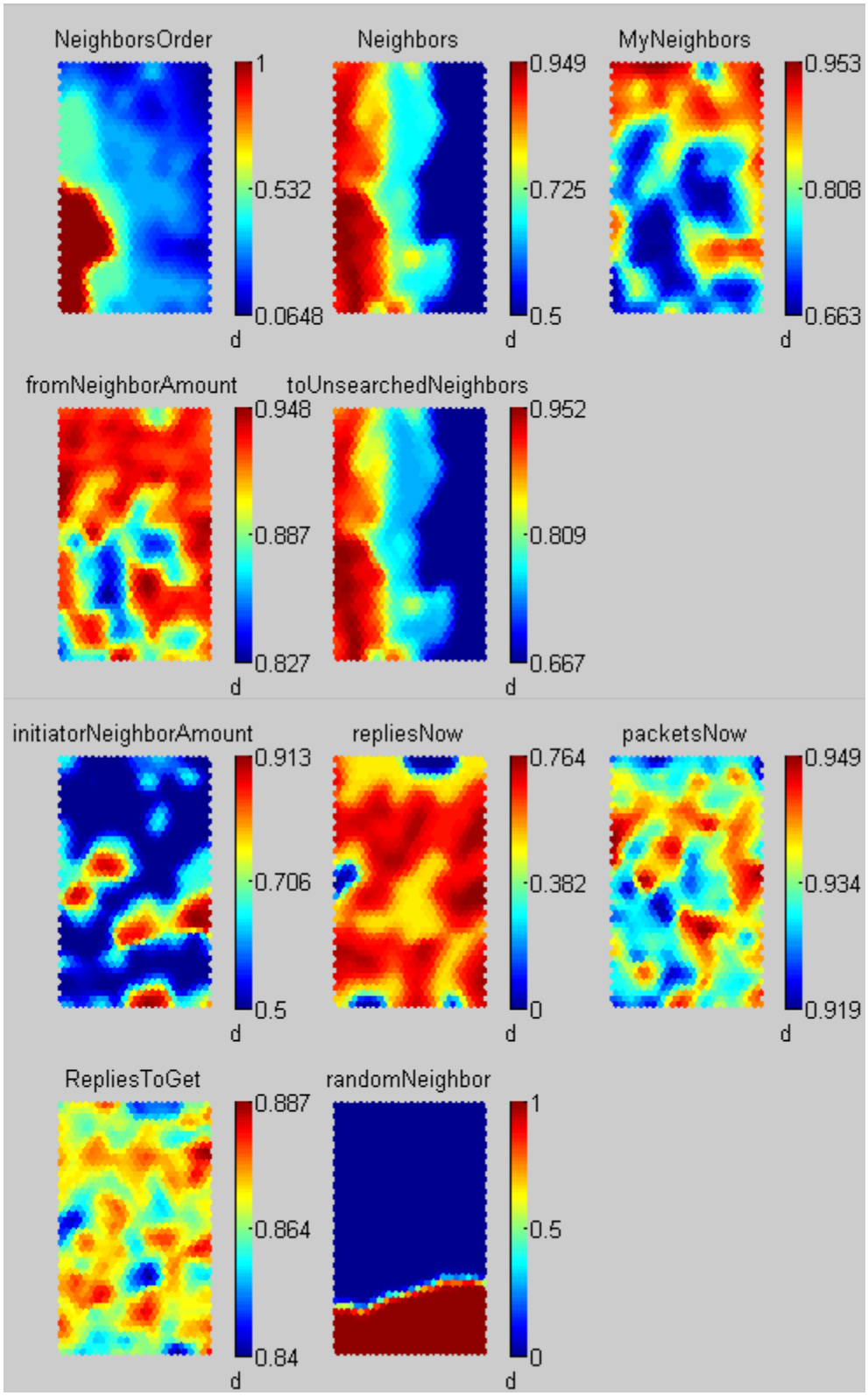

Figure 5.12 – Component plane for number of hops in the message is equal to 5

Analyzing component plane that is shown in Figure 5.12 one can see that NeuroSearch continues to avoid forwarding the queries to the most connected neighbors (where *NeighborsOrder* variable has high values). We are able to say that here we have quite complex situation with separation of different classes of decisions. Almost all variables take part into making of decisions. The classes that correspond to positive decisions about forwarding of the queries have low value of *initiatorNeighborAmount* variable, value of *MyNeighbors* variable should be high, value of *Neighbors* variable should be rather high than low and in the same time value of *NeighborsOrder* variable should be as low as possible and value of *randomNeighbor* variable should be equal to zero (in areas where variable *randomNeighbor* is equal to one we have very unclear structure and it is impossible to extract accurately decisions). Of course other variables also have some influence on decisions' mechanism of NeuroSearch, but this influence is not so big and it mostly duplicates contribution of other variables.

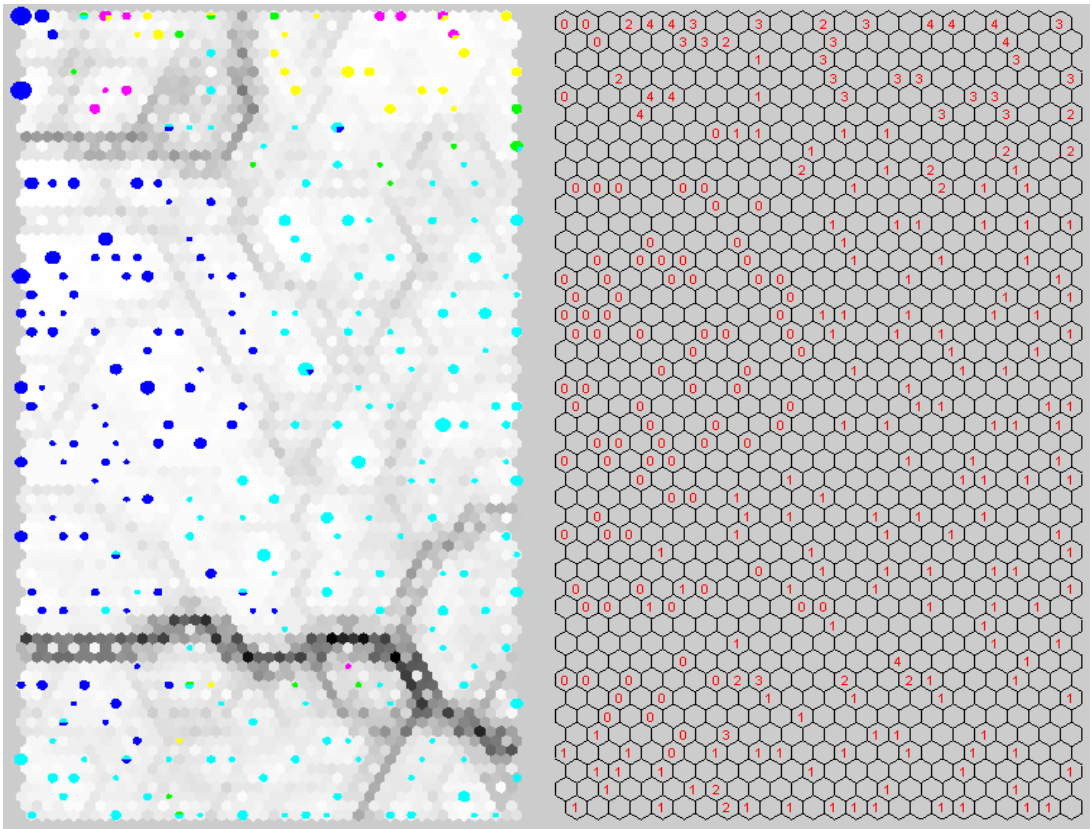

Figure 5.13 - U-matrix for number of hops in the message is equal to 6

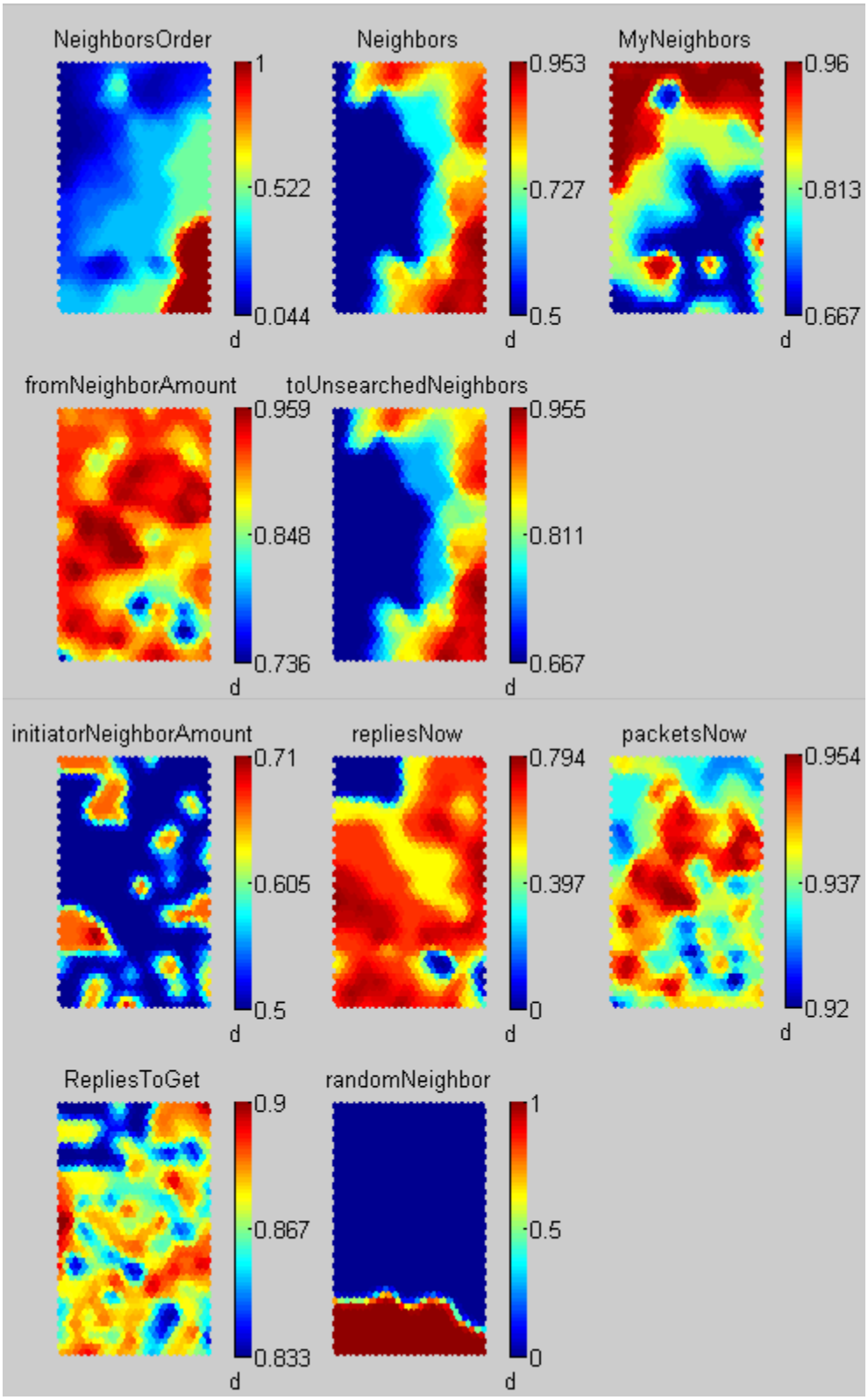

Figure 5.14 – Component plane for number of hops in the message is equal to 6

U-matrix and component plane for the hops value in the message is equal to 6 are shown in Figure 5.13 and Figure 5.14 respectively.

Analyzing U-matrix together with component plane one can see that there is better separation of decisions than in most previous cases. The positive decisions are located in the upper part of U-matrix. From this analysis and taking into account the previous steps we are able to say that NeuroSearch has now tendency to forward the queries to the middle-connected nodes. Because value of *Neighbors* variable is not so high and value of *NeighborsOrder* variable is small. Also positive decisions are characterized by the highest value of *MyNeighbors* variable. Influence of other variables is not so big.

Figure 5.15 illustrates U-matrix for the hops value in the message is equal to 7.

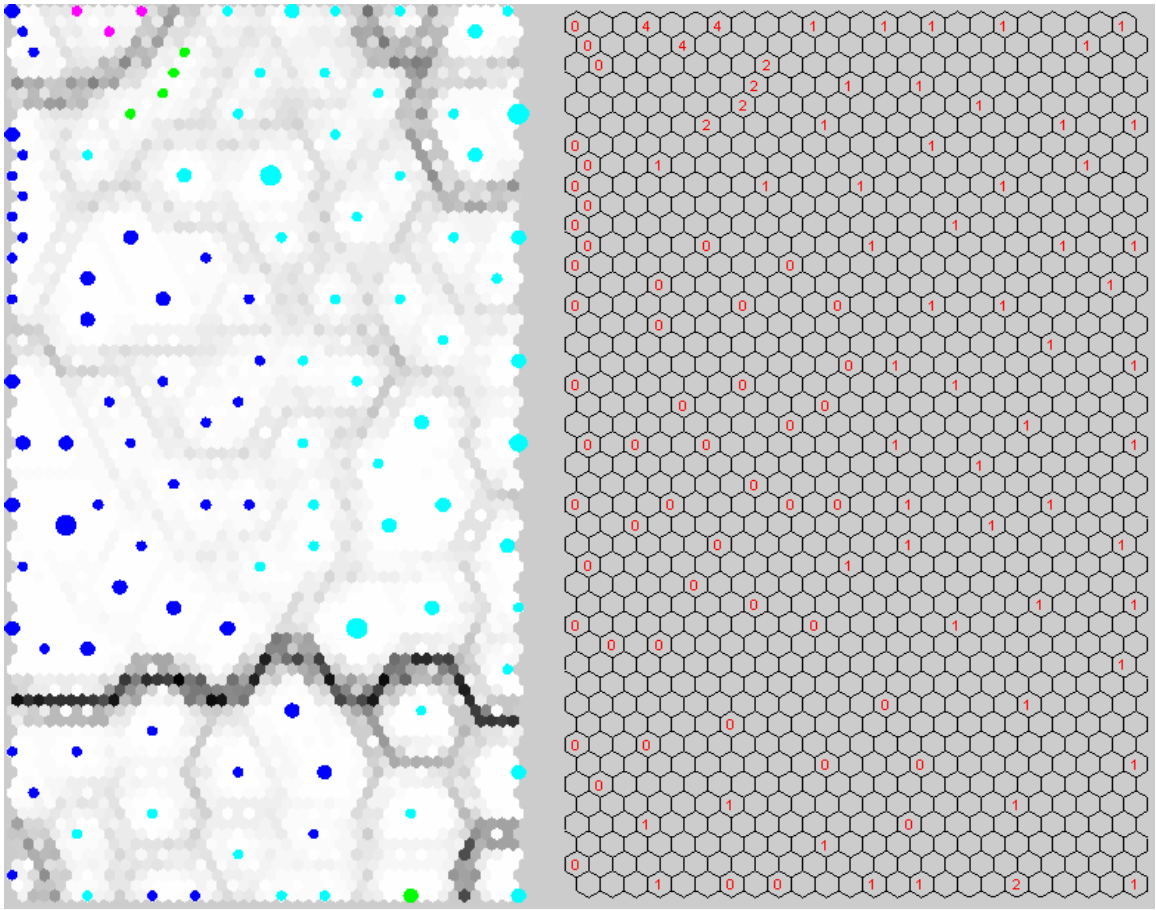

Figure 5.15 - U-matrix for number of hops in the message is equal to 7

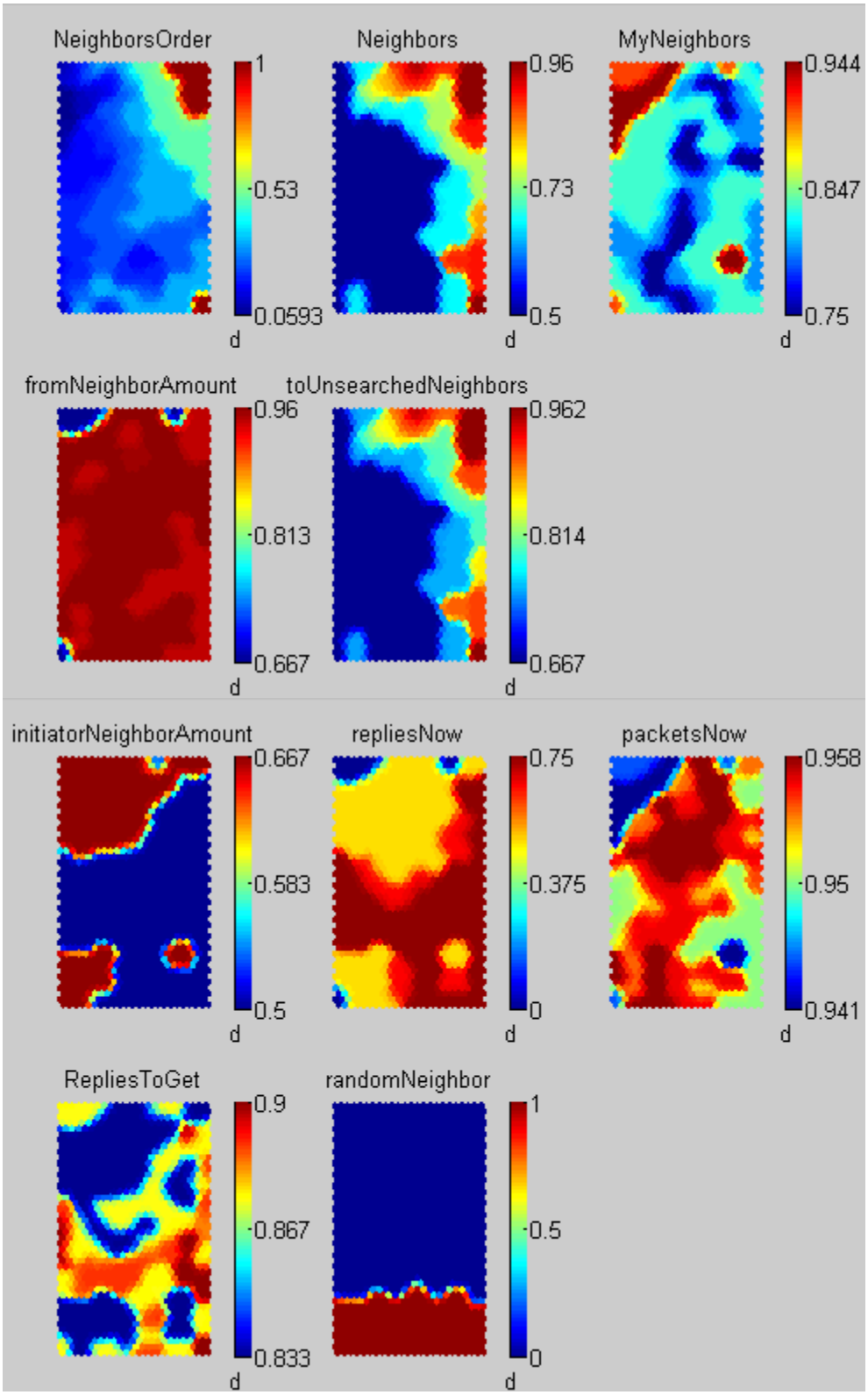

Figure 5.16 – Component plane for number of hops in the message is equal to 7

Analyzing U-matrix one can see that there are only few instances of classes that correspond to positive decisions about forwarding. All classes are well separable.

Component plane for case where number of hops in the message is equal to 7 is shown in Figure 5.16. Analyzing this component plane one can see that almost all variables give the same contribution to decisions. Good extraction of classes that respond to positive decisions about forwarding can be done using only two variables. For example we can select information from *fromNeighborsAmount* and *Neighbors* variables.

5.3 Rule Based Algorithm

Rule Based algorithm (RBA) was produced using analysis of SOM. RBA is based on rules which were extracted using U-matrix and corresponding component plane. To produce each rule we need to carefully investigate U-matrix together with decision distribution on it. After that we need to confront some areas where we want to determine a rule with ranges of variables in these particular areas. In other words we need merely to define more exactly dependencies that we found in section 5.2.

After careful analysis we got extracted the following set of rules:

1. if *Sent, currentVisited, From* or *toVisited* is equal to 1 then do not forward the query forward.
2. if *Hops = 1* and *NeighborsOrder = 1* then forward the query further.
3. if *Hops = 1* and *randomNeighbor = 1* and *MyNeighbors > repliesToGet* then forward the query further.
4. if *Hops = 0.5* and *fromNeighborAmount < 0.55* and *toUnsearchedNeighbors > 0.87* and *randomNeighbor = 0* then forward the query further.
5. if *Hops = 0.5* and *fromNeighborAmount < 0.68* and *toUnsearchedNeighbors > 0.8* and *randomNeighbor = 0* then forward the query further.

6. if *Hops = 0.5* and *fromNeighborAmount ≥ 0.7* and *toUnsearchedNeighbor ≥ 0.85* and *randomNeighbor = 0* then forward the query further.

7. if *Hops = 0.5* and *toUnsearchedNeighbors ≥ 0.83* and *randomNeighbor = 1* then forward the query further.

8. if *Hops = 0.5* and *NeighborsOrder ≥ 0.25* and *NeighborsOrder ≤ 0.35 and toUnsearchedNeighbors ≥ 0.75* and *toUnsearchedNeighbors ≤ 0.8* and *randomNeighbor = 0* and *repliesNow = 0* then forward the query further.

9. if *Hops = 0.5* and *NeeighborsOrder > 0.49* then forward the query further.

10. if *Hops < 0.34* and *Hops > 0.33* and *toUnsearchedNeighbors ≥ 0.66* and *toUnsearchedNeighbors ≤ 0.81* and *randomNeighbor = 0* and *NeighborOrder ≥* 0.25 then forward the query further.

11. if *Hops < 0.34* and *Hops > 0.33* and *toUnsearchedNeighbors > 0.81* and *NeighborsOrder ≤ 0.9* then forward the query further.

12. if *Hops < 0.34* and *Hops > 0.33* and *packetsNow < 0.8* and *NeighborOrder > 0.9* then forward the query further.

13. if *Hops > 0.3* and *Hops < 0.4* and *initiatorNeighborAmount < 0.67* and *repliesNow < 0.1* and *toUnsearchedNeighbor > 0.75* then forward the query further.

14. if *Hops > 0.3* and *Hops < 0.4* and *NeighborsOrder < 0.51* and *repliesNow > 0.1* and *toUnsearchedNeighbors > 0.79* and *initiatorNeighborAmount < 0.68* then forward the query further.

15. if *Hops = 0.25* and *toUnsearchedNeighbors ≥ 0.75* and *myNeighbors ≥ 0.8* and *NeighborsOrder ≤ 0.25* then forward the query further.

16. if *Hops = 0.25* and *repliesNow < 0.1* and *NeighborsOrder ≤ 0.35* and *toUnsearchedNeighbors > 0.74* and *myNeighbors > 0.74* then forward the query further.

17. if *Hops = 0.2* and *toUnsearchedNeighbors >0.75* and *toUnsearchedNeighbors < 0.93* and *myNeighbors > 0.88* and *NeighborsOrder < 0.26* then forward the query further.

18. if *Hops > 0.16* and *Hops < 0.18* and *NeighborsOrder < 0.17* and *toUnsearchedNeighbors > 0.79* and *toUnsearchedNeighbors < 0.86* and *initiatorNeighborAmount < 0.67* and *repliesNow < 0.1* then forward the query further.

19. if *Hops > 0.14* and *Hops < 0.15* and *toUnsearchedNeighbors = 0.75* and *fromNeighborAmount < 0.76* and *fromNeighborAmount > 0.66* and *myNeighborAmount = 0.9* then forward the query further.

One can notice all inputs in the rules are given after performing of corresponding scaling functions. If one is interested in real value of the parameters he can merely find these values by applying inverse scaling functions, in other words he should solve receiving equations.

After simulating NeuroSearch and RBA we got values of fitness function. These values are around 35000 for both algorithms.

Table 5.1 shows efficiency of four algorithms. One can see in this table that NeuroSearch and RBA have almost the same level of performance. This means that RBA adapted behavior of NeuroSearch and we are able to say that the SOM suits quite well for analyzing of NeuroSearch. Both these algorithms have better performance compared to BFS-2 and BFS-3 algorithms. Despite on that BFS-2 uses fewer packets than NeuroSearch does, but BFS-2 locates fewer resources. Also one can see that changing from BFS-2 to BFS-3 has disastrous consequences. We are able to say that NeuroSearch utilizes some quite good search strategy. It finds enough instances of available resources and in the same time uses satisfactory amount of packets compared to BFS-3.

Algorithm	Packets	Replies
BFS-2	3000	619
BFS-3	12464	1325
NeuroSearch	4703	979
RBA	4904	963

Table 5.1 – Comparison between algorithms

5.4 Discussion

After investigating behavior of NeuroSearch we are able to say that the SOM is an efficient tool to explore properties of data and make deep analysis of this data. We are unable to give exact answers about complexity of the system. Because from one side NeuroSearch has very simple decision mechanism, but from other side we have seen during our analysis that NeuroSearch has certain problems. These problems are mostly that opposite decisions are often near each other and it is impossible to 'extract' correctly knowledge about decisions without losing some of them. This fact definitely deserves our attention, because it is impossible to build good and stable system, which is based on unstable components. If we consider NeuroSearch without some problematic areas we are able to see quite explicit tendencies in its behavior.

The decisions are mostly based on very simple mechanism. It seems quite logical that NeuroSearch has different behavior when it starts the query and ends the query. Therefore in the current version of NeuroSearch parameter *Hops* plays very significant role. Also some parameters such as *Sent, currentVisited, From* and *toVisited* should stop further propagation of the query. Our analysis confirmed that it is almost true. There are only few points where it is not true. This might be caused by low-qualitative training, because

training algorithm might not have enough generations to find optimum solution. Positive decisions about forwarding are mostly based on two factors. These factors are number of receiver's neighbors (*toUnsearchedNeighbors* or *Neighbors*) and connectivity position of this receiver compared to other receiver (*NeighborsOrder*). It seems that other information is not so important for NeuroSearch and it is mostly used to improve accuracy of some decisions.

The search strategy of NeuroSearch is based on some tendencies. When NeuroSearch starts the query it looks for the most connected neighbors and sends the query to it. On the next steps NeuroSearch forwards the query only to several most connected nodes. This causes that in the middle phase of search process the queries are located at most connected nodes and this does not depend on the query's initiator. After that NeuroSearch starts to send query to nodes with high connectivity, but it avoids most connected nodes. NeuroSearch does not send the queries after 7 steps. This means that NeuroSearch represents itself mostly as some kind of BFS strategy.

Despite of that the SOM is quite powerful method for data analysis it requires some attention to output results. Because it's final performance highly depends on initial training parameters and characteristics of input data. Therefore results that were obtained from the SOM should be verified. That was done by producing and simulating RBA. RBA showed the same performance level with NeuroSearch, therefore obtained results are reliable.

6 Peer-to-Peer Simulation

Evaluation of new P2P protocols and other developed protocols or comparison between several existing protocols can be done either with help of live tests or using special simulation software [Carson 1997]. Making of live tests is problematic task because often necessary hardware is not available or some networks do not exist anymore like Gnutella network. Using simulation software is common way to perform such tasks. But simulation software has some disadvantages in comparison with live tests. We should define reliable scenarios, which should reflect real behavior of simulating system that is always problematic task. Errors on this stage can lead to disastrous consequences.

There is a lot of simulation software available on the market. Some of them are commercial products like QualNet [QualNet] and OPNET [OPNET] thus they are not available for our research. There are also freely distributed simulators like NS-2 [NS-2]. NS-2 is well documented software and can be easily extended for new protocol. Other free distributed software even if they are specially designed for P2P networking cannot be easily extended to provide required functionality for example supporting dynamical changes in the network. It is because this software is not well documented and some of them originally were designed for other tasks such as simulation of Distributed Hash Tables (DHT) [Rowstron & Drushel 2001]. Thus we chose to use NS-2 in our research, which provides enough functionality to build P2P protocols and network scenarios.

6.1 NS-2

This section is dedicated to review of characteristics of NS-2 simulator. It also introduces P2P extension for NS-2.

6.1.1 Concept Overview

Network Simulator NS-2 is discrete event based open-source simulator that was developed at UC Berkeley. NS-2 was developed for Unix-based environments, but also can be run under Windows OS using Cygwin [Cygwin]. NS-2 is able to perform the following networking simulations:

- Mobile networking (different MAC layer protocols, MobileIP, different types of routing agents: DSDV, AODV, TORA, DSR)
- Satellite networking
- Local area networking
- Wireless Sensor networking
- Transport layer protocols (UDP, TCP)
- Multicast and unicast routing algorithms
- Queue management mechanism (WFQ, RED, CBQ)
- Different radio propagation models (free space model, shadowing model, two-ray ground reflection model)

There are a lot of extensions available for NS-2, such as network animator, different topology generators and even distributed parallel version of NS-2 [PDNS]. NS-2 was originally developed as a part of REAL network simulator [REAL]. Currently NS-2 is a part of the VINT [VINT] project [NS-2]. VINT is assisting in protocols design, their evaluation and comparison. NS-2 is written in C++ and OTcl. It uses two different languages for different purposes. Naturally, it requires some system language (C++) for implementing different algorithms and manipulating data. But for configuration scenarios and fast tuning of the parameters we need some script language (OTcl). OTcl works significantly slower than C++ because OTcl is interpreted language, but code in OTcl can be changed very quickly and interactively. Also there are some ways to bind variables between these languages and invoke procedures from one language to another. There is also a suggestion in [NS-2] to write first implementation of new protocols entirely in OTcl and only after testing series rewrite it in C++ thus OTcl can be used for testing purposes. Simple schema that describes work with NS-2 is shown in Figure 6.1.

OTcl script defines main steps for simulation such as initialization of scheduler (start and end of simulation, at which time nodes have to send packets, link failure and so on), topology of network, format of results, and initialization of agents. Of course all these steps can be done from C++ code, but in this case we will lose flexible possibilities to tune necessary parameters and test quite big number of scenarios. There is tight connection

between C++ and OTcl. After running the simulation, Tcl interpreter configures the required C++ objects and invokes the necessary methods.

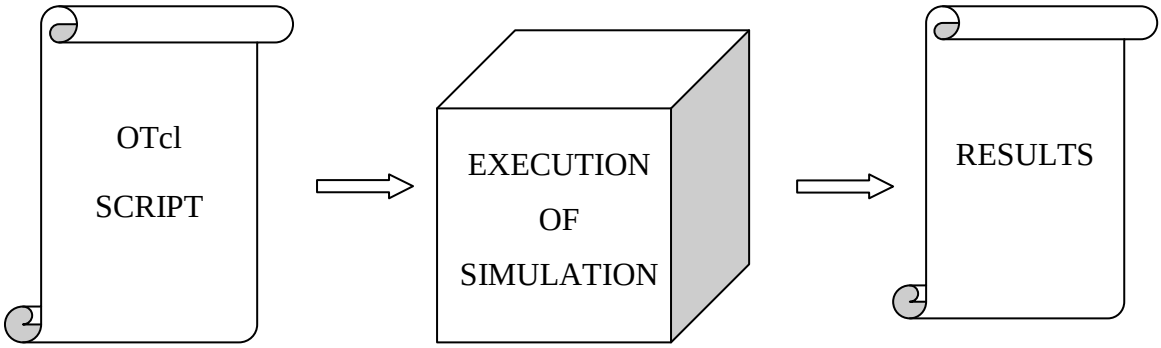

Figure 6.1 – Schema of work with NS-2

After completing the simulation there are several ways available to investigate the results using special tool called Network Animator (NAM) or using trace files. NAM is useful tool for finding errors in the implementation of a new developed protocol; also NAM can be used for demonstrating purposes. But when simulation scenario has a lot of nodes it becomes totally useless, because scenario cannot be illustrated entirely on a screen.

6.1.2 Discrete Event Scheduler

NS-2 as was mentioned in section 6.1.1 is event based simulator thus all simulation events should be defined. For this purpose *Scheduler* object is used. Figure 6.2 shows interaction between network objects and scheduler.

NS-2 supports two different types of scheduling: real-time and non-real-time. The non-real-time scheduler is set as default. The real-time scheduler can be used with a real network.

Each object that can interact with *Scheduler*, for instance such as *Simulator* object, has member functions to set simulation time of performing some actions during the simulation.

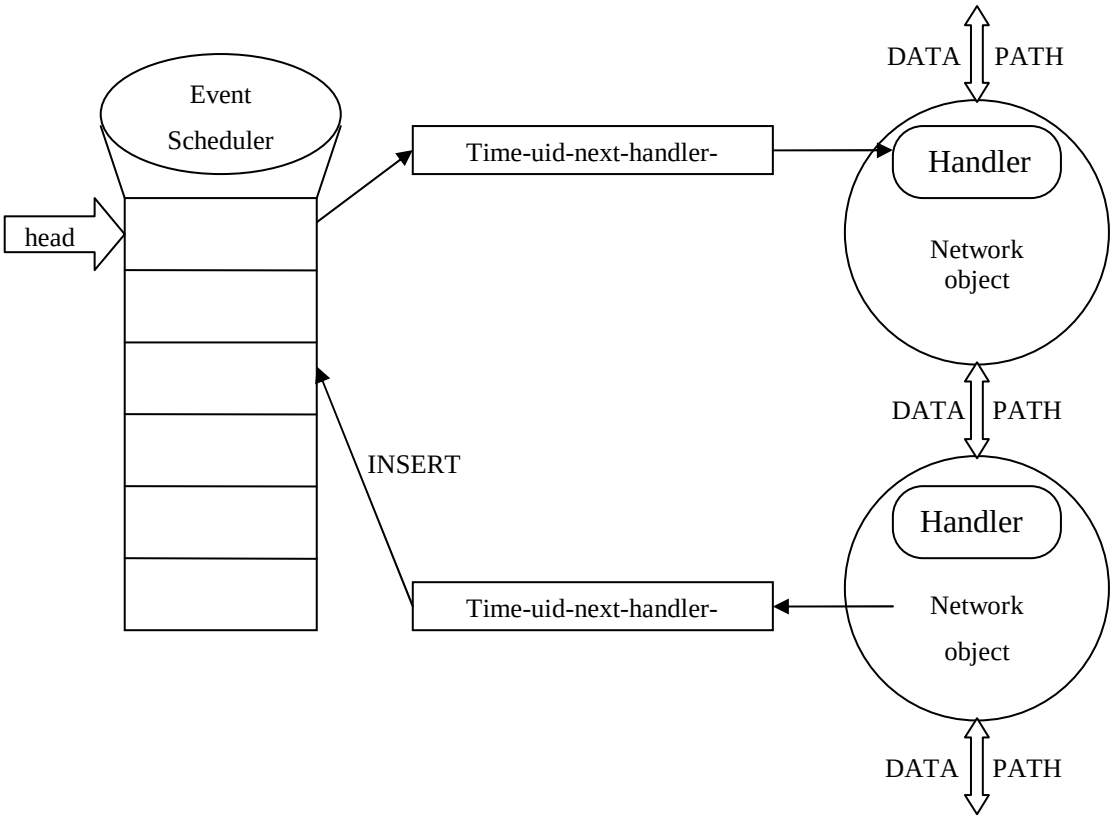

Figure 6.2 – Discrete Event Scheduler

There are several implementations of such functions:

- at arguments – schedule execution of code at specified time
- at-now arguments – schedule execution of code at now
- after n arguments – schedule execution of code after n seconds
- run arguments – start scheduler
- halt – stop scheduler

6.1.3 Network Components

In this section we describe basic network components such as node and link. They are small bricks for building any network topology.

A node is compound object composed of a node entry object and classifiers. Structure of each node is shown on figure 6.3.

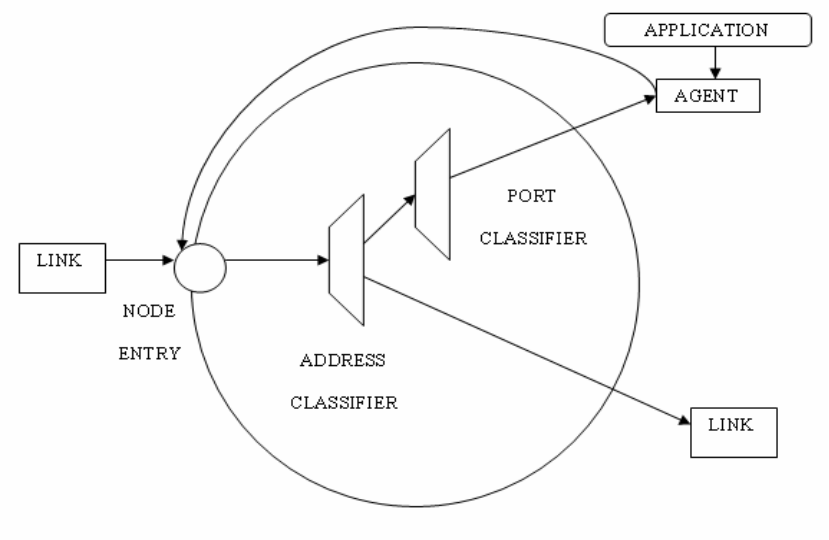

Figure 6.3 – Structure of node

Each node consists of two TclObjects: an address classifier and a port classifier. These classifiers distribute incoming packets to the correct agent or outgoing link. Each node has the following components:

- an address
- a list of neighbors
- a list of agents
- a node type identifier
- a routing module

Naturally, all nodes should be connected. For these purposes *Link* objects are used. A typical structure of the link is shown on figure 6.4. Queue implements management mechanisms for incoming packets. Delay models the link delay and bandwidth characteristics. *TTLChecker* (in figure 6.4 depicted as TTL) decrements the TTL field in each packet that it receives.

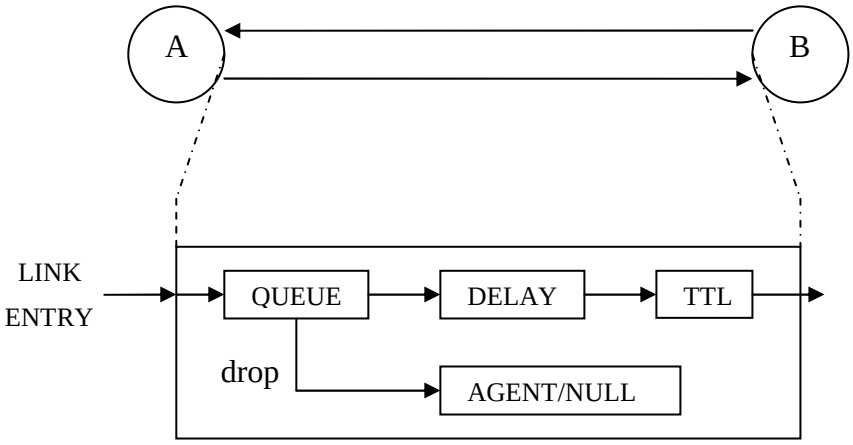

Figure 6.4 – Structure of link

6.2 P2P simulation on NS-2

Nowadays there is no good P2P simulator with full set of features for dynamic simulation. Some of them don't work in dynamic environment, others have unclear structure and it is difficult to extend them. NS-2 is powerful tool with full set of capabilities to provide qualitative simulation, but in author's knowledge there is no good implementation of P2P simulator for NS-2. In this section stages of development of P2P simulator for NS-2 are presented.

6.2.1 P2P Simulator Architecture

To develop Peer-to-Peer extension for NS-2 new P2P agent was created and some modules (simulator and node) were extended. Architecture of the application is presented in figure 6.5.

The application consists of three major components:

- P2P agent
- Extension of the node class
- Extension of the simulator class

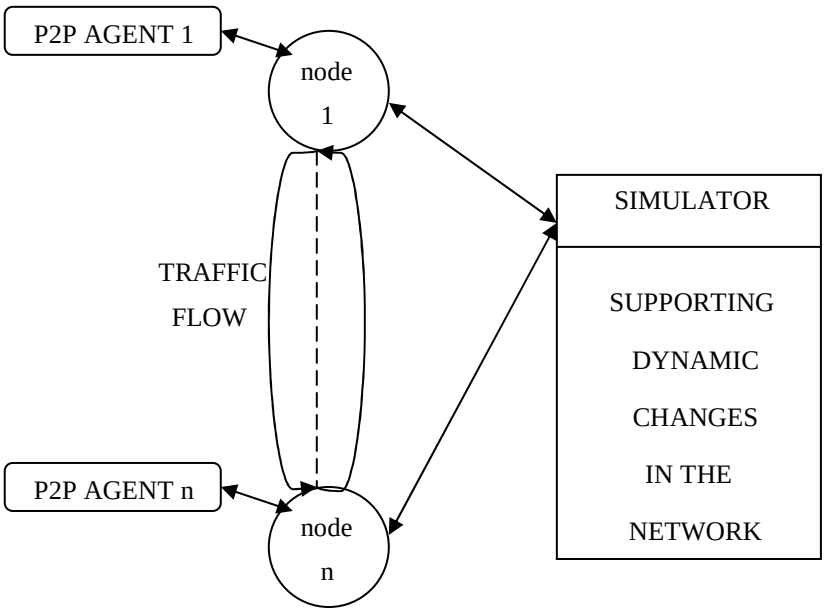

Figure 6.5 – Architecture of the application

Each P2P agent is attached to corresponding node and acts on application level of OSI model. Each agent has receiving and sending procedures. Sending procedure initializes each packet and sends it to the environment. Receiving procedure is used for processing received packets. This procedure determines the type of each packet and processes it depending on its type. On each step this procedure decreases TTL parameter and if this parameter is bigger than 0 it sends queries to all neighbors excepting the previous sender and nodes that already have been visited. For queries this procedure checks for the queried resource; if current node has the queried resource it sends back success reply using the same traveling path and if it has not, it sends back “normal” reply. Also this procedure monitors statistics and saves it to files. In addition to receiving and sending procedures, each agent has the following procedures:

- *Rndgenerate*
- *Getitem*
- *Shares*
- *Recneighbors*
- *Neighborhood*

Procedure *Rndgenerate* is used for accessing OTcl random generator. Procedure *Getitem* is used to determine querying resource. Procedure *Shares* is used for putting resources to each node. *Recneighbors* is used to determine number of neighbors of each receiver. Procedure *Neighborhood* is used to determine number of neighbors of current node.

Structure of the header of each packet has the following fields:

- *idsrc* - identifier of originator of query or reply
- *rcvd_from* - is identifier of neighbor that forwarded this packet to current node
- *hops* - array that consist of information about the path of each packet
- *target* - the name of queried resource
- *TTL* - time-to-live field of each packet
- *queryid* - unique identifier of each packet
- *packtype* - shows the kind of packet that agent processes (0 - query, 1 - success reply, 2 - normal reply)

The P2P agent can be easily extended for adding new capabilities, such as supporting of new protocols and redesigned version of old protocols.

Class *node* was extended to store different simulation parameters. This class can be easily accessed by each agent. Also other modules such as simulator can interact with class *node*. It is difficult to pass information directly from simulator module to each agent. Thus class *node* can be used as universal buffer to pass information between modules. The following parameters were added to class *node*:

- *int sharing[100]* – array containing shared instances of resources
- *int lastneighbors[100]* – array containing names of neighboring nodes of disappearing node
- *int status* – parameter shows in which mode the processing node is (1 - online, 0 - offline)
- *int cnvitycounter* – used for storing number of links for concrete node

- *double time_of_death* – time of disappearing of the node
- *double time_of_resurrection* – time of return of the node to the simulation environment
- *double total_time_of_life* – total calculated time of the presence of the node in the simulation environment
- *int ready* – parameter shows participation status of processing node in the simulation at particular period of time
- *int item* – identifier of queried item

Class *simulator* was extended to support and provide dynamical changes in P2P environment. NS2 is event based simulator, in other words all events such as appearance and disappearance of nodes should be defined before starting the simulation. So we needed a support for dynamical changes in real time simulation. Also this class was extended to provide a support for starting preparation of all nodes and automatic sending of queries. Extended class *simulator* can process the following key procedures invoked from OTcl script:

- *Nodesprepare*
- *Ready*
- *Startquery*
- *Rndgenerate*
- *Neighborkill*
- *Checkneighbors*
- *Kill*

Procedure *Nodesprepare* is used for initialization of parameters for each node (*status*, *ready*, *cnvitycounter*). Procedure Ready is used to determine which nodes should act as initiators of queries at particular moment. Procedure *Startquery* is used for generating start queries from defined nodes. Procedure *Rndgenerate* is used for generating random numbers using OTcl library. Procedure *Neighborkill* is used for saving information about neighbors of disappearing node. This information will be used for future reappearing of the

node. Procedure *Checkneighbors* is used to find out information about status of neighbors. Procedure *Kill* is the main procedure for supporting dynamic changes in the P2P network. This procedure defines the following actions:

- Monitoring status of all nodes
- Determining probability for staying, returning and removing of nodes
- Processing and saving statistics to files

Probability of appearing and disappearing of each node depends on connectivity of each node. Recent studies [Ripeanu 2002] have shown that in Peer-to-Peer networks some nodes have more stable behavior than others. These nodes usually are accommodated in the central parts of the network and have more connections than others.

6.2.2 Usage of P2P Simulator for NS-2

To start the simulation OTcl script has to be used. In this script the following key commands are used:

- *set n [$ns node]*
- *$ns duplex-link $n0 $n1 1Mb 10ms DropTail*
- *set p0 [new Agent/Neurosearch]*
- *$ns attach-agent $n0 $p0*
- *$ns preparing_nodes*
- *$p0 shares 18*
- *$ns startquery*
- *$ns at 1000 "finish"*
- *$ns run*

Command *set n [$ns node]* is used to initialize each node. Command *duplex-link* is used to provide duplex link between two nodes (for above example between nodes n0 and n1, bandwidth is 1Mb, delay is 10ms and queue mechanism is Drop Tail). Command *set p* is used for the initialization of each agent. Command *$ns attach-agent $n0 $p0* is used to attach p0 agent to node n0. Command *$ns preparing_nodes* is used for the initialization of

dynamic properties of each node. Command *$p0 shares 18* means that node p0 has resource with id=18. Command *$ns startquery* is used for automatic starting of queries. Command *$ns at 1000 "finish"* defines the time of simulation (1000 sec). Command *$ns run* starts the simulation.

The final results of simulation are saved to files in the same directory where OTcl script is stored. File *query.txt* contains simulation information related to spread of the queries through the network. Format of the *query.txt* file is shown in table 6.1. Information about successful replies is stored in *reply.txt* file. Format of the *reply.txt* file is shown in table 6.2. To decrease time of simulation it is possible to use only necessary short information about location of the resources. For this purpose *f_reply.txt* file is used. Format of the *f_reply.txt* file is shown in table 6.3.

Node's ID	Packet's ID	Number of node's neighbors	Receiver's ID	Number of Receiver's neighbors	TTL	Query Initiator's ID	PATH

Table 6.1 – Format of 'query.txt' file

Node's ID	Packet's ID	Node's status	Previous hop's ID	ID of resource's location	Resource's ID	TTL	TIME

Table 6.2 – Format of 'reply.txt' file

Node's ID	Packet's ID	TTL	ID of resource's location	PATH

Table 6.3 – Format of 'f_reply.txt' file

Node's ID	Packet's ID	Node's status	Previous hop's ID	Resource's ID	TIME

Table 6.4 – Format of 'normal.txt' file

If query did not find target resource at particular node, this node sends back reply to notify about this. Information about such replies is stored in *normal.txt* file. Table 6.4 contains format of *normal.txt* file. Meanings of most table fields of tables are intuitively

understandable. There are only several fields with non-clear meaning. Field PATH contains list of nodes that packet has visited during its travel in the network. Field Node's status shows status of current node (offline/online). Field TIME stores time of occurring of particular event, for example time of locating some resource at particular node.

List of queried resources are stored in file *targets.txt*. Statistics about the appearance and disappearance of nodes are stored in *kill.txt* and *resurect.txt* files correspondingly. There is also possible to investigate results of simulation with help of NAM.

6.2.3 RBA Extension

Having implemented basic P2P simulator on NS-2 we can easily extend it for a new P2P algorithm. In our case we extended it to support RBA, which was described in chapter number five. Results of RBA simulation are described in chapter number seven. Usage of RBA extension from user's point of view is the same with that was described in the previous section. One difference is that now user needs to define desired simulation mode. This can be done by using command *$ns set_algorithm RBA.* If user does not need to use RBA then he merely does not include this command to OTcl script. Results of RBA simulation can be shown on the screen or put to the output file.

7 Open Problems

Despite the fact that NeuroSearch shows quite good performance it has some problems like every new algorithm. One problem is that it is difficult to explain behavior of NeuroSearch. This problem has been solved in chapter number five using SOM. Another problem is that NeuroSearch is based on evolutionary computing and thus it needs a lot of training time to adjust weights of the MLP. Also the behavior of NeuroSearch in distributed and dynamic P2P environment is unknown. The benefits of using NeuroSearch in static environment is not so valuable, because it is well known that in real life situations P2P networks are characterized by high dynamicity rate. Therefore it is very important to know about behavior of NeuroSearch in dynamic environment. In this chapter we mostly pay attention to the above described problems.

7.1 NeuroSearch in Dynamic Environment

Since RBA that was described in chapter number five is based on decision mechanism of NeuroSearch it is possible to evaluate behavior of NeuroSearch using RBA in dynamic environment.

As a simulation environment P2P extension for NS-2 described in chapter number 6 was used. For the test case we used topology, which is the same with topology used in chapter number 5 to test RBA in static P2P environment. This topology obeys power-law distribution (2) where γ for Barabasi-Albert graph [Barabasi & Albert 1999] is 3. The procedure, which provides dynamical changes to the P2P environment, has quite simple structure. All nodes have two probabilities that are used together to leave and join the network. The first class of probabilities is defined randomly before starting the simulation. Its purpose is to determine stability of each node. The second class of probabilities is determined during the simulation according to nodes' connectivity with neighboring nodes using formulas (33) and (34).

$$leaving\ \ probability = \frac{1}{connectivity} \tag{33}$$

$$joining\ \ probability = \frac{1}{26 - connectivity} \tag{34}$$

Constant 26 in formula (34) was selected to guarantee non-zero value in the dominator, since maximum connectivity is 25. The purpose of the second class of probabilities is to guarantee stability of highly-connected nodes according to [Ripeanu 2002]. If some node has to start its query and this node is in offline mode, the node returns to the environment. Also if all neighbors of some node left the network then this node automatically leaves the network. Since we need to evaluate results from all nodes we selected simulation time to be equal to 100 seconds. During this time all nodes one by one send their queries with interval of 1 second.

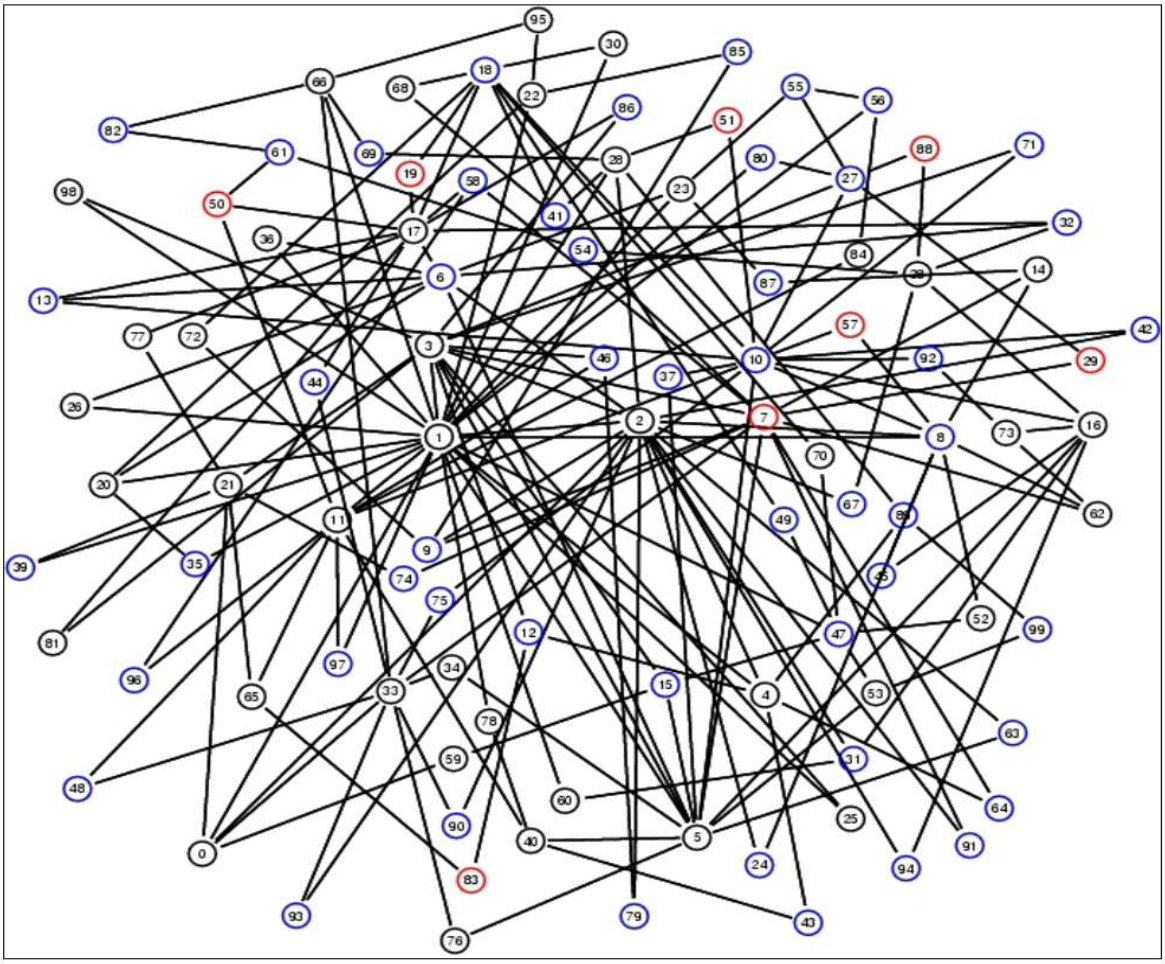

Figure 7.1 Topology with dynamical changes

To provide quite high dynamicity to the environment we invoked procedure, which is responsible for dynamical changes 4 times per second. Figure 7.1 illustrates the topology after 20 seconds of simulation. In the figure black nodes denote nodes, which have stayed in the environment all the time, red nodes are currently in offline mode and blue node returned to the environment at this moment. Also from the figure one can see that in general most of high-connected nodes show high stability and most low-connected nodes are characterized by high dynamicity as was defined by our scenario.

To make qualitative evaluation of performance RBA was compared to BFS-2 and BFS-3 algorithms in static and dynamic environments. Number of replies and amount of used packets in static environment are shown in Figure 7.2 and Figure 7.3 respectively.

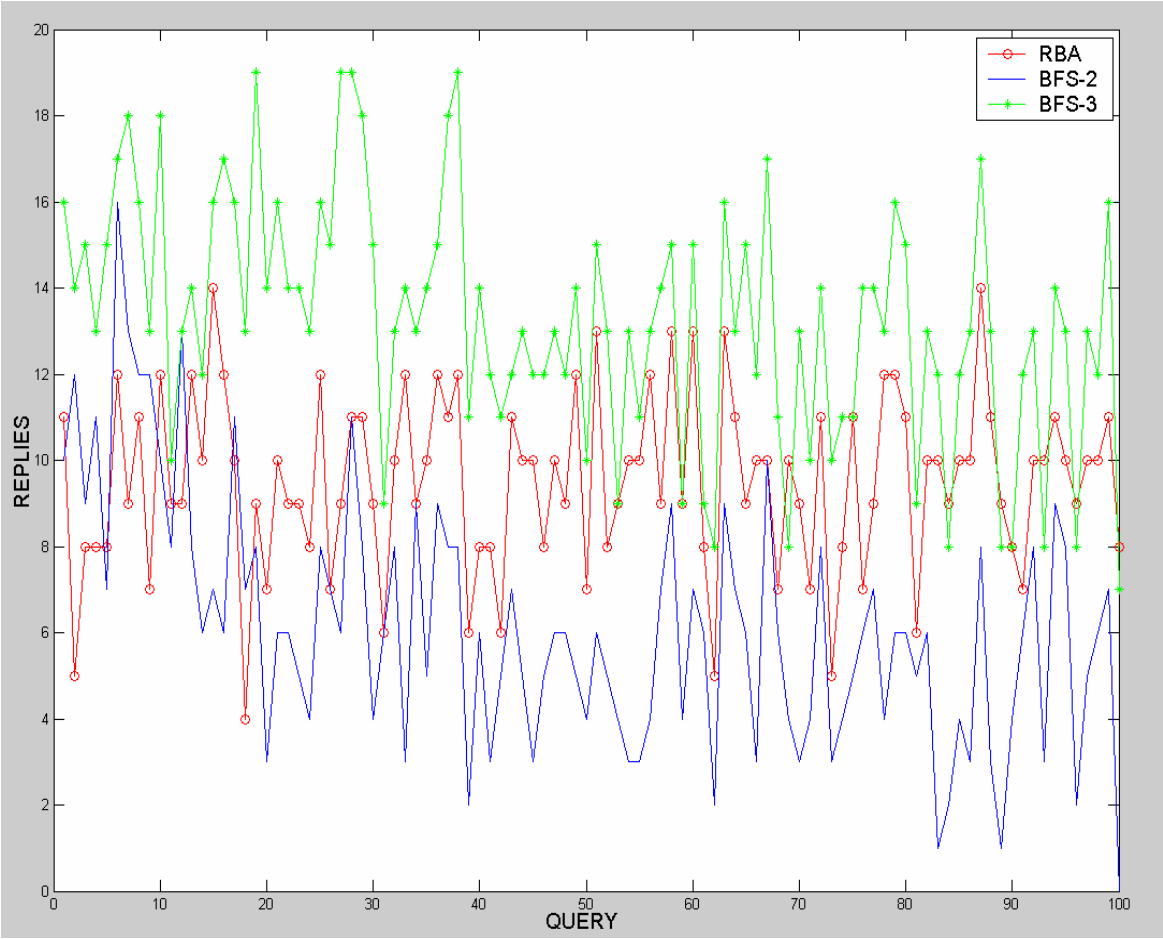

Figure 7.2 – Replies located by the algorithms in the static environment

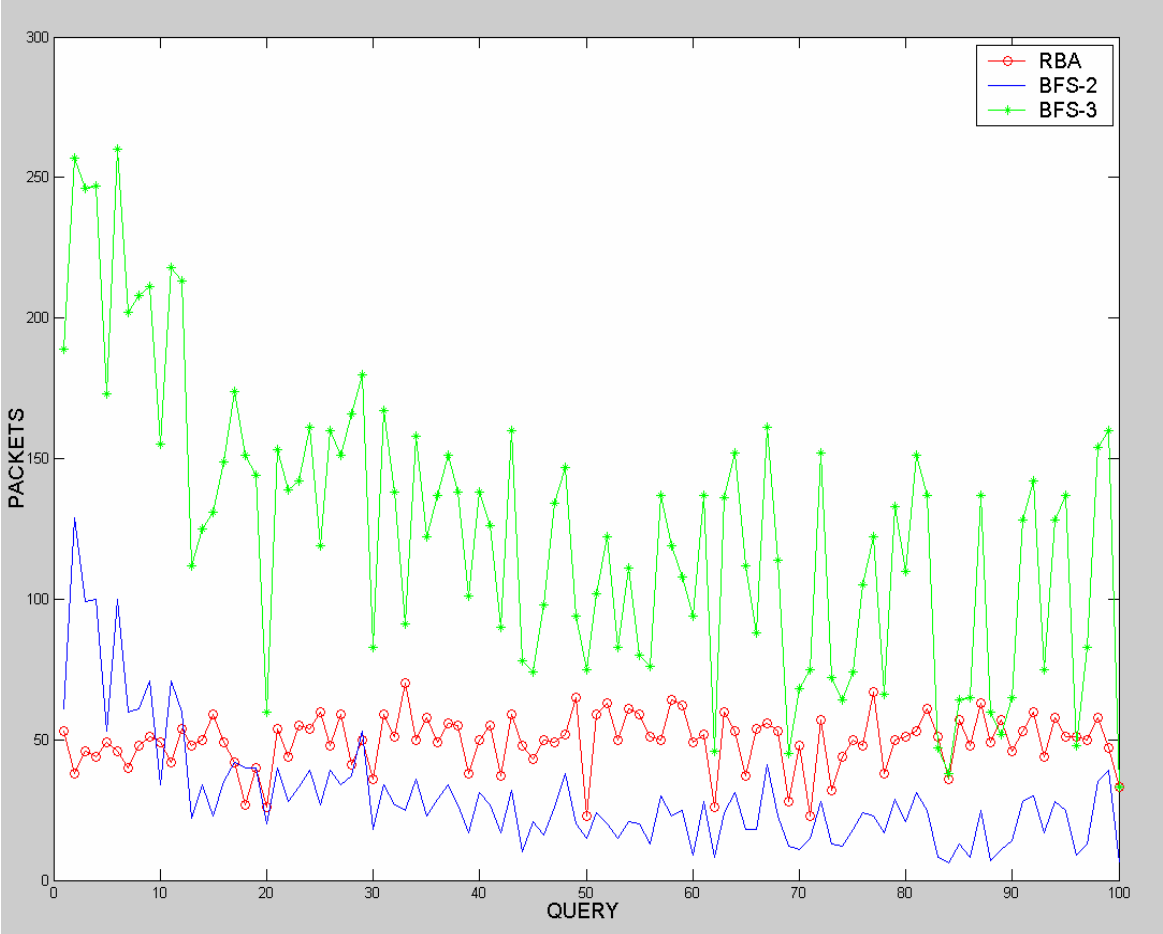

Figure 7.3 – Packets used by the algorithms in the static environment

Total number of located resources and used packets in static environment can be found in table 5.1. Analyzing Figure 7.2 one can see that mostly RBA locates more resources than BFS-2 algorithm and in the same time significantly less than BFS-3 algorithm. From Figure 7.3 one can see that in general RBA uses more packets than BFS-2 algorithm and significantly less than BFS-3 algorithm. Mostly in points where RBA located less resources than BFS-2 algorithm, RBA used less packets than BFS-2 algorithm. In terms of located resources this situation satisfies us because RBA is based on NeuroSearch's decision mechanism that is trained to locate only half of available resources. In some points RBA locates more resources than BFS-3 algorithm and in the same time uses less packets. This means that if some resource is not common in the network, RBA and as consequence NeuroSearch is able to find enough instances of this resource.

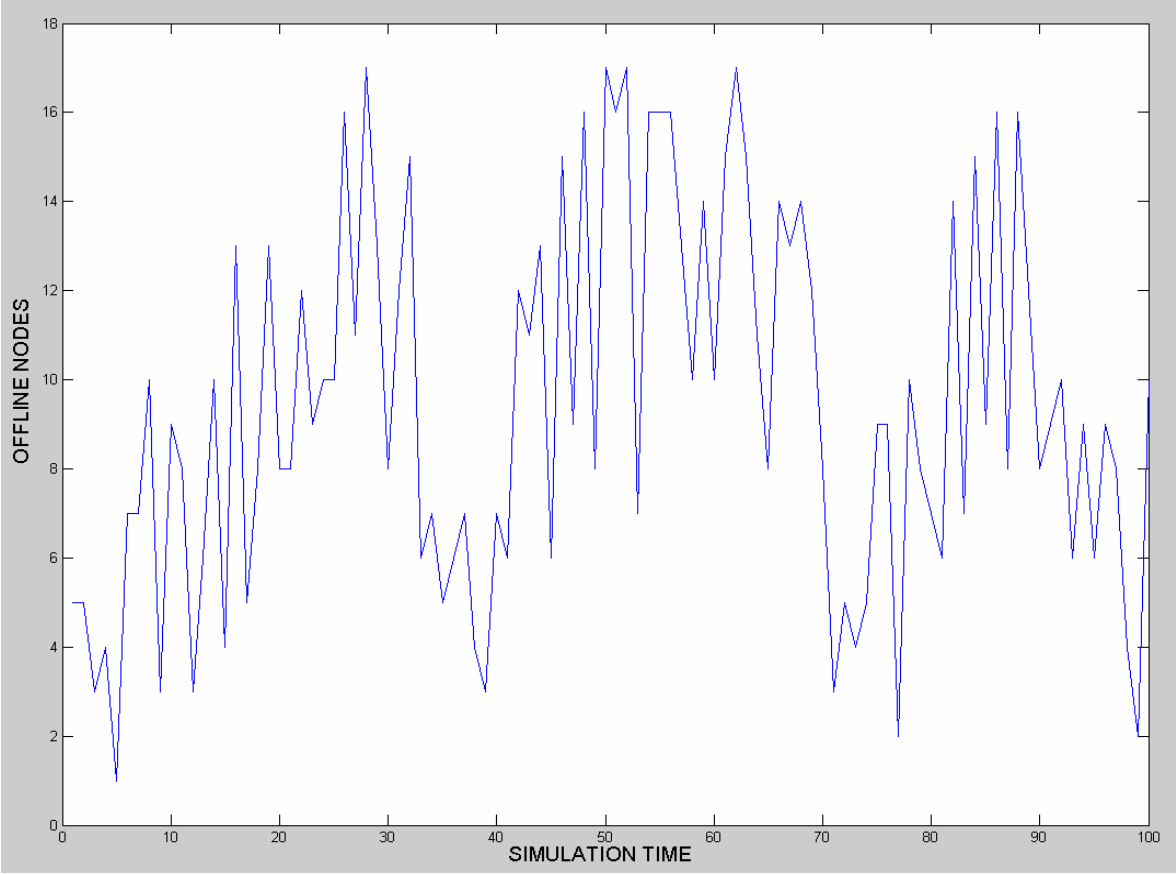

Figure 7.4 – Dynamical changes in the network

Figure 7.4 illustrates dynamical changes in the network. In Figure 7.4 one can see the amount of nodes in offline mode during the simulation. This figure shows that simulation conditions provide to the network rapid topological changes during the simulation time.

Number of replies and amount of used packets in the dynamic environment are shown in Figure 7.5 and Figure 7.6 respectively. Analyzing these figures one can see that performance of the algorithms did not suffer so much in the dynamic environment. The algorithms are able to locate satisfying amount of resources. The behavior of these algorithms is quite similar to their behavior in the static environment. Still BFS-2 uses the least amount of packets among the compared algorithms. But it finds in general like in the static environment the least instances of available resources. BFS-3 finds more resources but it uses significantly more packets than other algorithms. Performance of RBA is also very similar with its performance in the static environment.

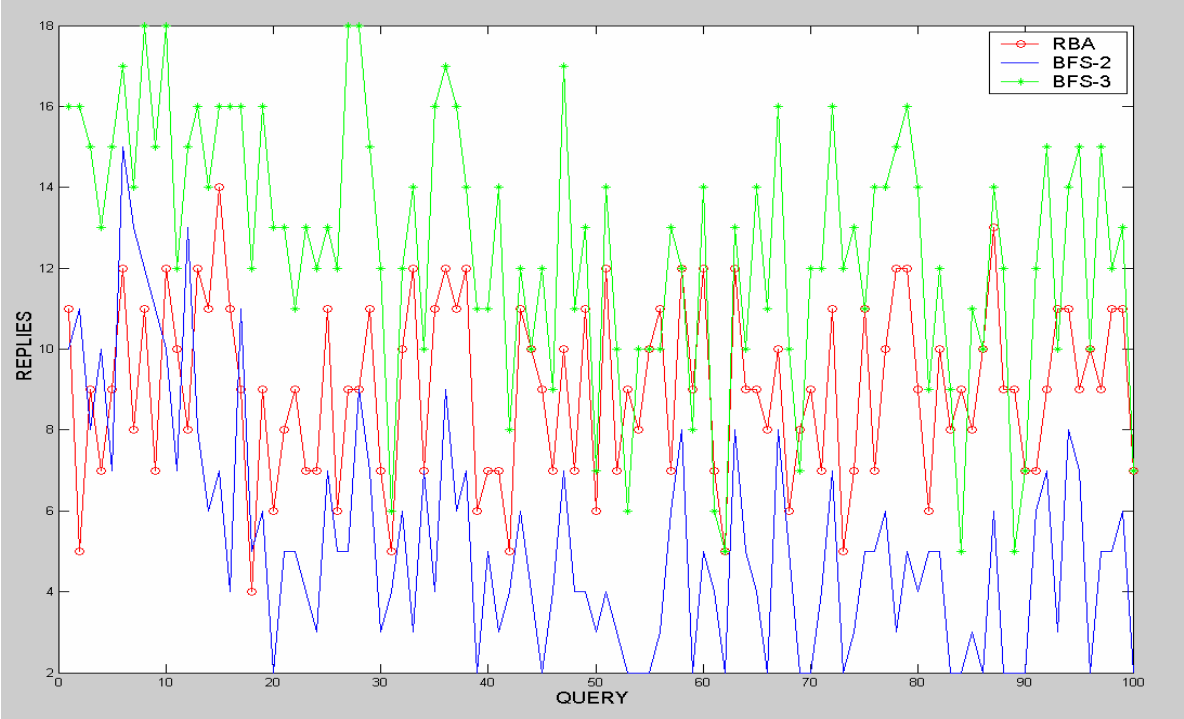

Figure 7.5 - Replies located by the algorithms in the dynamic environment

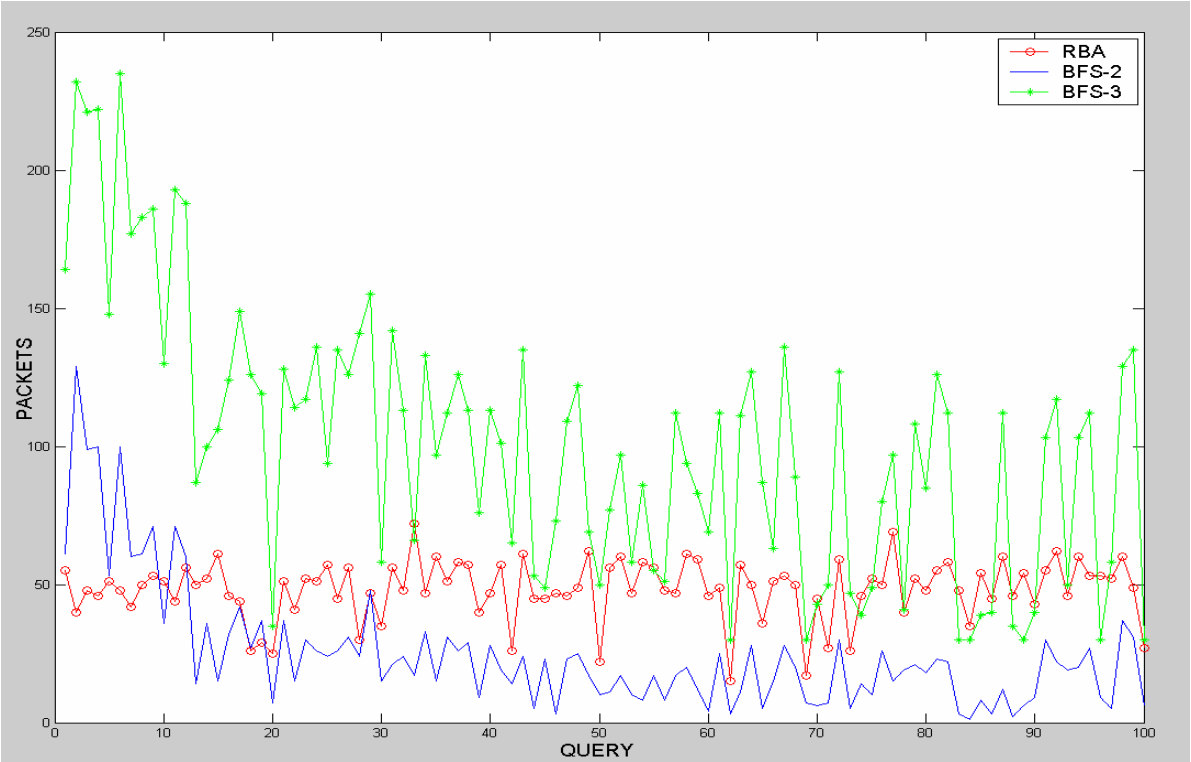

Figure 7.6 - Packets used by the algorithms in the dynamic environment

Total number of located resources and used packets in the dynamic environment can be found in table 7.1.

Algorithm	Packets	Replies
BFS-2	2515	528
BFS-3	10040	1245
RBA	4865	900

Table 7.1 – Comparison between algorithms in the dynamic environment

Analyzing results that were obtained from the static environment (Table 5.1) and the dynamic environment (Table 7.1) one can see that in general total performance of the compared algorithms has not been changed so much. They still are able to find enough resources in the dynamic environment. There are two possible causes, which can explain the fact that all investigated algorithms found a little bit fewer resources in dynamic environment. The first cause is that some nodes in offline mode can contain queried resources. Therefore these resources cannot be located from the network. The second cause is that some nodes in offline mode even if they do not contain queried resources might lie on possible path of the query. This is more related to BFS type of algorithm, because if this happened in the beginning phase of the query path this definitely limits the further propagation of the query. Naturally, one can see from Table 7.1 that all algorithms used less packets in the dynamic environment than in the static environment. BFS-3 algorithm got especially good benefit from decreasing connectivity between nodes and the size of the environment. Disappearing of some nodes did not affect so much on amount of located resources and in the same time decreased number of used packets. This coheres with theoretical background of BFS-strategies in high-connected environments such as power-law environments: in more connected networks we have bigger number of packets.

7.2 Optimization Problems

In this section we consider problems, which are related to training of the MLP for resource discovery problem. The main question that arises here is whether it is possible to provide efficient training algorithm for decision mechanism (in our case MLP) of NeuroSearch, which at the same time will spend less time on training than Evolutionary Algorithm (EA). One of possible solutions is to use some kind of BP method. But to apply BP we need to know exact relationship between input information and desired output information. Great advantage of EA is that we do not need such relationships; in working process of EA it can find desired relationships by itself in unsupervised manner.

One of possible ways to get desired input-output relationships is to use some deterministic search strategy to get some heuristic equations. For example these equations can be based on success rate and amount of used packets. Two conditions should be satisfied to guarantee the existence of solution. The first condition is that there must be one-for-one kind of dependency between input and output pairs. The second condition is that data must be well separable, because we do not want a system where number of neurons tends to infinity.

It is easy to prove that it is impossible to satisfy the first condition. Assume that we have arbitrary power-law graph with some resource distribution. Consider some simple situation when we need to make decision about further forwarding that is based on some success rate. Figure 7.7 illustrates simple sub graph with the above-described situation.

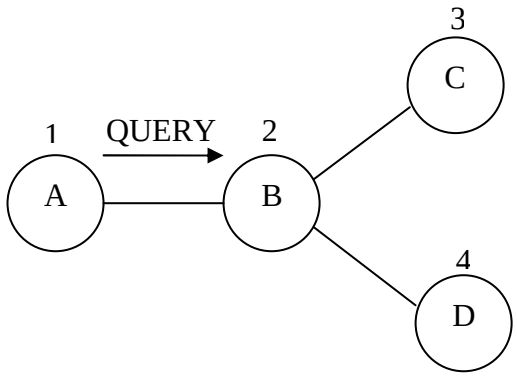

Figure 7.7 – Simple subgraph

From figure 7.7 one can see that it is impossible to make clear decision about further forwarding that is based on some success rate from node 1 to node 2. Because if target of the query is one of B, C or D resources then NeuroSearch should forward the query, but if target is A resource then NeuroSearch should not forward the query further. Since all available resources can be queried we come to controversial situation. Of course by analyzing Figure 7.7 one can say that it is possible to make decision that is based on some probability. Since we can find three resources for four queries we can forward the query with probability 0.75. But it is true only for this simple situation. In real life scenarios we are dealing with much complex situations where we have huge graphs with large clustering coefficient. This is leading us to situation where all directions are almost equal to each other. At the same time we need to guarantee good generalization properties of the algorithm. This also adds some additional complexity to our situation because we need to use some independent environment together with learning environment. In other words we need some joint decisions that work quite well in both environments. Therefore it is hard to make clear decision that is based on probability in such complex environment.

Satisfying of the second condition that was described in the beginning of this section is also increasing complexity of the training problem. One of possible ways to do this is to use some clustering method to investigate complexity of the resource discovery problem. This can be done for new algorithm in the same way that was described in chapter number five.

Having read the above described problems one can think that it is impossible to implement some BP based training algorithm for resource discovery problem. But it might not be so. The main problem is how to collect and process statistical information from P2P environment. During working on the thesis it has been investigated that using BFS strategy as basis for new algorithm is not so good approach. To investigate this data from P2P environment was collected using P2P simulator described in chapter number six. This data contained statistics, which was based on dependency between input parameters of NeuroSearch and some success rate. After that special software was written to process collected information and based on this information desired outputs for the input data were associated. Unfortunately after that data contained a lot of controversial points, e.g. points

that provide different output with the same inputs. All attempts to process it or remove all controversial points by changing them to their average value were unsuccessful. Average value was quite the same for different input information. It was impossible to distinguish the data. In contrast to this, NeuroSearch (based on EA) does not have such problems. It is because during the training NeuroSearch automatically and in implicitly manner utilizes topological properties of the environment. Eventually NeuroSearch after selecting a general strategy (it can be easily seen by analyzing evolution figure in [Vapa et al. 2004] where performance of the algorithm becomes quite high in early generations) starts only to refine this general strategy. On one hand it helps to avoid problems with controversial outputs, but on the other hand this has some hidden reefs, because algorithm might converge in a local minimum.

Thus we need some algorithm that can provide good searching strategy from first steps of its work to collect data for BP training. In the same time this algorithm during the work can collect data that does not have controversial points. It is hard to propose right algorithm for this case. One of possible solutions can be based on Steiner tree [Gilbert & Polak 1968] algorithm. But nevertheless data obtained from this algorithm should be checked for controversial points and separation. If it is not possible to avoid controversial points then simple management mechanism is needed that can provide general decisions for controversial points. This task will be much easier than providing the same approach for BFS strategy. It is because we do not have a lot of variants for further searching of resources. We merely have some general strategies. In other words it is possible to say that we can reduce dimension of the task. Separation abilities of the data can be checked using clustering analysis.

Another interesting problem that is related to optimization process is the definition of the fitness function. Indeed due to it we can judge about performance of the algorithm and in the same time the algorithm uses it to adjust its behavior. The fitness function is defined with some quite big weight constant to ensure that algorithm is able to find half of resources. But this might lead us to some problems. It is because selection mechanism works to choose only the best individuals. Thus from initial steps of optimization process we are selecting individuals that are able to find more resources. The easiest way to find

resources is to use some kind of BFS strategy. Therefore the algorithm might stick in local minimum from the beginning of the optimization, because all selected individuals are quite the same and obey BFS strategy. One possible solution is to use much smaller constant instead of 50. This constant has to be selected taking into account average path length. It is because using average path length we can reach near half of all nodes without any problems and therefore this means that we are able to find needed resource with high probability. This should provide more fair competition between found resources and used packets. As a consequence our population will contain more heterogeneous individuals.

7.3 Discussion

As we have seen, behavior of RBA and as a consequence behavior of NeuroSearch is quite stable under dynamic conditions. This is quite natural because during the simulation the topology does not loose its power-law properties. In other words migration of the nodes in the core of the network has fewer ratios than in the edges.

Despite that the difference between BFS-2 algorithm and BFS-3 algorithm is only 1 step in terms of hops, BFS-2 is unable to locate enough resources when its travel path goes through low-connected nodes. Using of BFS-3 is also not good idea, because we should pay expensive price using disastrous amount of packets. All these evidence show that using BFS strategy cannot provide successful solution to the problem. NeuroSearch algorithm combines properties of BFS and DFS strategies. This makes NeuroSearch unique in the sense of searching strategy. On one side this provides fast location of enough amount of resources and on the other side avoids flooding problems. Analyzing the simulation results of all investigated algorithms from the dynamic and the static environment we are able to conclude that BFS strategy is very sensitive to the size of the network. BFS-3 algorithm used significantly less packets in the dynamic environment where size of the P2P network was a little less than in the static environment all the simulation time. RBA used approximately the same amount of packets in both environment and therefore we are able to say that it is not strongly sensitive to size of the network as BFS-3 algorithm. Also we can forecast that if size of the network is increasing a little then we will have significantly more disastrous consequences from using BFS strategy.

Another problem that was described in this chapter is related to optimization. Despite of the fact that using GA is good approach, which allows finding the solution, it is time consuming process. The next step of my work is aiming to provide supervised training which will allow finding the solution much faster. In this chapter we paid attention to problems that have to be solved to provide supervision to the training process.

8 CONCLUSIONS

The thesis was aimed to solve optimization problem of NeuroSearch in dynamic environment. To do this we need answers to some questions that lie on the way to solution of the optimization problem in dynamic environment. First we needed to know more about decision mechanism of NeuroSearch algorithm. This was done in chapter number five using the Self-Organizing Maps.

After extracting the rules that describe behavior of NeuroSearch we produced RBA, which is based on these rules. Benefits from RBA can be seen from two positions. Knowing the decision mechanism we are able to use its dependencies to produce rules more efficiently. In other words if we decide to implement some alternative optimization mechanism, for example those that was described in chapter number seven, we will be able to make some needed corrections to the optimization. Also it was shown that RBA and NeuroSearch algorithms are quite similar. Thus we investigated behavior of RBA and as consequence behavior of NeuroSearch in dynamic environment. The results of the simulation have shown that performance of the investigated algorithms did not suffer so much in dynamic environment. More detailed information about results of the simulation can be found in chapter number seven.

Thus in the thesis we have partially solved preliminary tasks that lie on the way to finding good optimization method that can be used to train NeuroSearch in dynamic environment. Now obtained results are opening straight way to the goal. Also the thesis leaved some open issues for future work, because it is not possible to cover big amount of existing problems and at the same time solve them. But anyway we are now able to produce the closest goals for the future work. First, in future works, we need to decide which algorithm is most suitable choice to play role of supervised base in training process. One of possible choices can be some modification of Steiner Tree algorithm. To do this some data mining techniques like the SOM can be applied. Also some basic management system is needed to avoid controversial situation, which surely will be presented in decision mechanism. And only after careful investigation and testing of the algorithm we will be able to judge about feasibility of applying it to our problem.

References

[Adamic et al. 2001] Adamic L.A., Lukose R.M., Puniyani A.R., Huberman B.A., Search in power-law networks, Physical Review E 64, 2001.

[Adamic et al. 2002] Adamic L.A., Lukose R.M., Huberman B.A., Local search in unstructured networks, arXiv:cond-mat/0204181, 2002

[Albert & Barabasi 2001] Albert R., Barabasi A.-L., Statistical mechanics of Complex Networks, arXiv:cond-mat/0106096 v1, 2001

[Ata et al. 2003] Ata S., Gotoh Y., Murata M., Replication strategies in Peer-to-Peer services over power-law overlay networks, Proc. APNOMS, 2003

[Barabasi & Albert 1999] Barabasi A. L., Albert R., Emergence of scaling in random networks, Science 286, 509-512, 1999

[Barabasi 2002] Barabasi A. L., Linked: The New Science of Networks, Perseus Publishing, 2002

[Bishop 1995] Bishop C., Neural networks for pattern recognition, Clarendon Press, Oxford, 1995

[Blickle & Thiele 1995] Blickle T., Thiele L., A mathematical analysis of tournament selection, Proc. of the 6th International Conference on Genetic Algorithms, 1995

[Carson 1997] Carson M., Application and protocol testing through network emulation, Internetworking Technologies Group NIST, 1997

[Cheese] Cheese Factory project website, http://tisu.it.jyu.fi/cheesefactory/

[Chen & Smith 1999] Chen S., Smith F. S., Putting the ”Genetics” back into Genetic Algorithms (Reconsidering the role of crossover in hybrid operators), The Robotics Institute Carnegie Mellon University, Pittsburgh, 1999

[Cygwin] Cygwin website, http://www.cygwin.com/

[eDonkey] eDonkey website, http://edonkey.com/

[Edutella] Edutella website, http://edutella.jxta.org/

[Eryurek & Upadhyaya 1990] Eryurek E., Upadhyaya B. R., Sensor validation for power plants using adaptive back propagation neural networks, IEEE Trans. on nuclear science, vol 37:2, 1990

[Faloutsos et al. 1999] Faloutsos M., Faloutsos P., Faloutsos C., On power-law relationships of the Internet topology, Proc. SIGCOMM, 1999

[Freenet] Freenet website, http://freenet.sourceforge.net/

[Gilbert & Pollak 1968] Gilbert E.N., Pollak H.O., Steiner minimal trees, SIAM Applied math, 16:1-29, 1968

[Gnutella] Gnutella website, http://gnutella.wego.com/

[Groove] Groove website, http://www.groove.com/

[Hebb 1949] Hebb D.O., The organization of behavior: A neurophysiologic theory, John Wiley and Sons, New York, 1949

[Hecht-Nielsen 1987] Hecht-Nielsen R., Kolmogorov's mapping neural network existence theorem, IEEE, in Proceedings of the International Conference on Neural Networks, 1987

[Holland 1975] Holland J., Adaptation in natural and artificial systems, the University of Michigan Press, Ann Arbor, 1975

[Iamnitchi et al. 2002] Iamnitchi A., Ripeanu M., Foster I., Locating data in (Small-world?) Peer-To-Peer Scientific Collaborations, Proc. 1st International Workshop on Peer-To-Peer Systems, 2002

[Kazaa] Kazaa website, http://kazaa.com/

[Kaski et al. 1999] Kaski S., Venna J., Kohonen T., Coloring that reveals high dimensional structures in data, Proc 6th International Conference on neural information processing, vol 2:729-734, 1999

[Kim et al. 2002] Kim B.J., Yoon C.N., Han S.K., Jeong H., Path finding strategies in scale-free networks, Physical Review E 65, 2002

[Kohonen 1993] Kohonen T., Things you haven't heard about the Self-Organizing Map, IEEE international conference on neural networks, vol 3:1147-1156, 1993

[Kohonen 2001] Kohonen T., The Self-Organizing Maps, Springer, Berlin, 2001

[Krogh & Hertz 1995] Krogh A., Hertz J. A., A simple weight decay can improve generalization, In advances in neural information processing systems, vol 4:950-957, Morgan Kauffmann Publishers, San Mateo CA, 1995

[Lv et al. 2002] Lv Q., Cao P., Cohen E., Li K., Shenker S., Search and replication in unstructured peer-to-peer networks, ACM Press, 2002

[Matlab] Matlab website, http://www.mathworks.com/

[McEliece et al. 1987] McEliece R. J., Posner E. C., Rodemich E. R., Venkatesh S. S., The capacity of Hopfield associative memory, IEEE Trans. on info series, vol 33:461-482, 1987

[NS-2] NS-2 website, http://www.isi.edu/nsnam/ns/

[OPNET] OPNET website, http://opnet.com/

[Oram 2001] Oram A., Peer-to-Peer, O'reilly & Associates Inc, 2001

[PDNS] PDNS (Parallel Distributed Network Simulator) website, http://www.cc.gatech.edu/computing/compass/pdns/

[QualNet] QualNet website, http://scalable-networks.com/

[REAL] REAL network simulator website, http://www.cs.cornell.edu/skeshav/real/

[Ripeanu 2002] Ripeanu M., Peer-to-Peer architecture case study: Gnutella network, Proc. of 1st International Workshop on Peer-To-Peer Systems, 2002

[Rowstron & Drushel 2001] Antony I. T. Rowstron, Drushel P., Storage management and caching in PAST, a large scale, persistent peer-to-peer storage utility, In Symposium on Operating Systems Principles, 2001

[SETI@Home] SETI@Home at Berkeley website, http://setiathome.ssl.berkeley.edu/

[Schollmeier 2001] Schollmeier R., A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications, Proc. of Peer-to-Peer Computing, 2001

[SOM toolbox] SOM toolbox website, http://www.cis.hut.fi/projects/somtoolbox/

[Stoica et al. 2002] Stoica I., Adkins D., Zhuang S., Shenker, S., Surana S., Internet Indirection Infrastructure, Proc. SIGCOMM, 2002

[Vapa et al. 2004] Vapa M., Kotilainen N., Auvinen A., Töyrylä J., Hyytiälä H., Vuori J., NeuroSearch: evolutionary neural network resource discovery algorithm for peer-to-peer networks, University of Jyväskylä, 2004

[Vesanto & Alhoniemi 2000] Vesanto J., Alhoniemi E., Clustering of the Self-Organizing Maps, IEEE trans. on neural networks, vol 11:3, 2000

[Vesanto et al. 2000] Vesanto J., Himberg J., Alhoniemi E., Parhankangas J., SOM Toolbox for Matlab 5, Helsinki University of Technology, Report A57, 2000

[VINT] VINT (Virtual InterNetwork Testbed) website, http://www.isi.edu/nsnam/vint/

[Wiener 1948] Wiener N., Cybernetics or control and communication in the animal and the machine, John Wiley & Sons, 1948

[Widrow & Lehr. 1990] Widrow B., Lehr M. A., 30 years of adaptive neural networks: perceptron, madaline, and back propagation, Proceedings of the IEEE, vol 78:9, 1990

[Yamauchi & Randall 1994] Yamauchi B. M., Randall B. D., Sequential behavior and learning in evolved dynamical neural networks, Technical Report CES-93-25, Dept of Computer Engineering and Science, Case Western Reserve University, Cleveland, 1994

[Yang & Garcia-Molina. 2002] Yang B., Garcia-Molina H., Improving search in peer-to-peer networks, Proc. 22nd International Conference on Distributed Computing Systems, 2002