Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Cybernetics

BACHELOR THESIS

Visualisation and Analysis of Artificial Neural Network Behaviour

Prague, 2012 **Pavel Fencl**

Declaration

I hereby declare that I have completed this thesis independently and that I have used only the sources (literature, software, etc.) listed in the enclosed bibliography.

In Prague on 28.5.2012 .....

Acknowledgement

I would like to thank everybody who helped me with completing this thesis, mainly to my supervisor Prof. Assoc. Pavel Nahodil for help and experienced advices through formally part of my work and to my consultant Ing. Jaroslav Vítků for his time, significant help and guidance throughout the entire process of development and research.

Abstrakt

Moje bakalářská práce zkoumá problém umělých neuronových sítí z hlediska jejich chování. Rozvíjí více než třináctiletý výzkum přístupu k umělému životu, vedený Pavlem Nahodilem na ČVUT v Praze, fakultě kybernetiky.

Mým cílem bylo vytvořit nástroj, umožňující lepší pochopení, demonstraci a zkoumání vnitřních zákonitostí chování neuronové sítě třetí generace. Výsledkem je aplikace v jazyce Java vystavěný na knihovně vyvinuté univerzitou v Marylandu podporující kreslení planárních grafů. Pro práci s daty a strukturou grafu jsem využil poznatky získané během studia z oblasti grafů, kybernetiky a umělé inteligence.

Program umožňuje uživateli mnoho možností interakce s animací sítě či jejím rozložením. Mnou navržená aplikace je schopna zobrazit síť a umožňuje vizualizaci jejího chování v čase.

Kromě funkční aplikace naprogramované v jazyce Java je také výsledkem této práce několik navržených a implementovaných algoritmů, které jsem odzkoušel na různých topologiích neuronových sítí. V práci proto představím nejen samotnou aplikaci, ale i navržené algoritmy pro návrh rozložení sítě včetně diskuze užitečnosti jimi poskytovaných výsledků.

Abstract

My thesis explores a problem of neural networks with regard to their behaviour. It deepens a more than thirteen years old research of approach of artificial life conducted by Pavel Nahodil on CTU in Prague (Faculty of Cybernetics).

My objective was to develop a tool that would enable better understanding, demonstration and research of internal behavioural regularities of a third generation neural networks. The resulting application is written in Java and supports drawing of planar graphs. It is based on a library developed by University of Maryland. For manipulation with data and structure of the graph, I used knowledge acquired when studying the area of Graphs, Cybernetics and Artificial Inteligence.

Program gives the user various options in interaction with the network animation or distribution. My application is able to display the network and enables a visualisation of its behaviour in real time.

Aside from functioning application written in Java, this work also resulted in creation of several implemented algorithms tested on various topologies of neural networks. Thus, in my work, I will present not only the application itself, but also the algorithms for distribution of network and a discussion of usefullness of their results.

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra kybernetiky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: Pavel F e n c l

Studijní program: Kybernetika a robotika (bakalářský)

Obor: Robotika

Název tématu: Vizualizace a analýza chování umělé neuronové sítě

Pokyny pro vypracování:

1. Seznamte se se základy neuronových sítí třetí generace a metod vizualizace grafů.
2. Zvažte a navrhněte vhodné metody vizualizace topologie a chování těchto umělých neuronových sítí.
3. Navrhněte a implementujte aplikaci pro vizualizaci pomocí Vámi navržených metod.
4. Vyzkoušejte navržené druhy vizualizace pro různé příklady neuronových sítí.

Seznam odborné literatury: Dodá vedoucí práce.

Vedoucí bakalářské práce: doc. Ing. Pavel Nahodil, CSc.

Platnost zadání: do konce zimního semestru 2012/2013

L.S.

prof. Ing. Vladimír Mařík, DrSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 9. 1. 2012

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

BACHELOR PROJECT ASSIGNMENT

Student: Pavel F e n c l

Study programme: Cybernetics a Robotics

Specialisation: Robotics

Title of Bachelor Project: Visualisation and Analysis of Artificial Neural Network Behaviour

Guidelines:

1. Study the basics of artificial neural networks of 3rd generation and graph visualisation.
2. Design methods suitable for visualisation of topology and behaviour of these Artificial Neural Networks.
3. Design and implement application for visualisation of your designed methods.
4. Test the designed methods of visualisation for various examples of neural networks.

Bibliography/Sources: Will be provided by the supervisor.

Bachelor Project Supervisor: doc. Ing. Pavel Nahodil, CSc.

Valid until: the end of the winter semester of academic year 2012/2013

prof. Ing. Vladimír Mařík, DrSc.
Head of Department

L.S.

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, Januar 9, 2012

Contents

List of Abbreviations **ix**

List of Figures **x**

List of Algorithms **xii**

1 Introduction **1**

1.1 Thesis Outline 2

2 Theoretical Foundation **3**

2.1 Artificial Neural Networks 3

2.2 Graph Theory 5

3 State of the Art **7**

3.1 Source library 7

3.1.1 Requiements 7

3.1.2 Options 8

3.1.3 Conclusion 8

4 Application Design **9**

4.1 Objects Structure 11

4.2 Data Loading 12

4.2.1 Loading the topology data 12

4.2.2 Loading the spike train data 13

4.3 Topology Graphics 14

4.3.1 Drawing 15

4.3.2 Dynamics 16

4.4 Spiketrain Graphics 18

4.4.1 Drawing . . . 19
4.4.2 Dynamics . . . 20
4.5 Graphical User Interface . . . 21
4.5.1 File loading . . . 21
4.5.2 Tools menu . . . 22
4.5.3 Algorithms menu . . . 24

5 Algorithms and Experiments **27**
5.1 Easy Circle . . . 27
5.2 Computed Circle . . . 28
5.3 Basic Weight Algorithm . . . 32
5.4 Advanced Weight Algorithm . . . 33
5.5 Advanced Crossing Algorithm . . . 34
5.6 Signal Flow Algorithm . . . 35
5.7 Power . . . 38
5.8 Experiments . . . 39
5.8.1 Signal Flow on Moderate Network . . . 39
5.8.2 Power Advanced Algorithm Hiearchical Example . . . 40
5.8.3 Comparison of Active algorithms Small Network . . . 42
5.8.4 Comparison of Active algorithms Fullbinded Network . . . 44
5.8.5 XOR tested by algorithms . . . 45

6 Thesis Conclusion **48**

Bibliography **51**

A Contents of CD **I**

B User Manual and Implementation notes **II**
B.1 Launching the Experiments . . . II
B.2 Manual . . . II
B.3 Source Data Examples . . . IV

List of Abbreviations

ANN	Artificial Neural Network
SNN	Spiking Neural Network
P2D	Piccolo2D
GUI	Graphical User Interface
FPS	Frames Per Second
OSL	Open Source Library

List of Figures

2.1 Leaky Integrator model 4
2.2 Integration of Pulses 4
2.3 Spiketrain by Maass 5

4.1 Application Appearance 9
4.2 Network Basics 10
4.3 Network Example 16
4.4 Application Tools 17
4.5 Dynamics 18
4.6 Spiketrain Example 20
4.7 Graphical User Interface 21
4.8 Files Loading 22
4.9 Tools Menu 23
4.10 Algorithms Menu 25

5.1 Circle algorithms 29
5.2 Solve Matrix 30
5.3 Multi-neuron Computed Circle 32
5.4 Feedforward ANN 36
5.5 Signal Flow special states 37
5.6 Signal Flow on Hieararchical Example 39
5.7 Power Advanced Algorithm Hieararchical Example 40
5.8 Comparison of Active algorithms Small Network 42
5.9 Comparison of Active algorithms Fullbinded Network 44
5.10 XOR dynamic context 45
5.11 XOR tested by weight algorithms 46
5.12 XOR tested by power algorithms 47

B.1 Application Appearance III
B.2 Topology File IV
B.3 Spiketrain File IV

List of Algorithms

1 Computed Circle Algorithm 31
2 Step of Basic Weight Alhorithm 33
3 Step of Advanced Algorithm 34
4 Creating of Power Binds 38

Chapter 1

Introduction

For a long time, researchers of cybernetics dealt with a very complex problem. How to build the artificial agent that would be able to adapt to the complexity of the real world? The complexity and observability of the real environment is simply too high to be exactly desribed by anything less complex than environment itself (Vítků, J., 2011). Commonly agents are realizes simplified mathematical form of biological organism. Because of that, agent must be flexible, adaptive to environment, able to learn from unpredicted situations and reacts to them.

This thesis conscerns with the agents controlled by artificial neural networks. In department of cybernetics agents are developed to be used in diverse environments with high rate of success. In case of agents controlled by artificial neural networks, we are able to observe and measure their behaviour by analysis of their responses to various inputs, calculating the internal variables (e.g. the states of particular neurons) and by observing their consequent reactions. From these measurments we are able to get a vast amount of data by which we can describe behaviour of the agent and perceive agent as a black box. The main problem with use of artificial neural networks is in the fact that in most cases we are unable to understand the pronciples how it works inside. Therefore we often treat with the neural networks as a black-boxes.

My goal is to create a tool which enables us to at least partially look inside this black box and tell us how it probably works.

In order to do this, I implemented an application which places neurons onto a 2D plane based on some given criteria. By looking at the resulting graph, we should be able to understand the network more easily.

1.1 Thesis Outline

Here is a brief description of this thesis outline.

Chapter 1 The first chapter provides an introduction to this work and describes the main goals of Bachelor Thesis.

Chapter 2 The second chapter introduces the technical aspects of ANN and basic priniciples of graph theory..

Chapter 3 The third chapter provides information about the present state-of-the-art tools for visualising ANN and describes relationship of this work to the project of Mr. Nahodil's department.

Chapter 4 The fourth chapter describes functions of visualising tool, its internal functions and provides summary of user interface.

Chapter 5 The fifth chapter describes algorithms which were chosen and implemented.

Chapter 6 The last chapter contains conclusion and compares results with predicted goal.

Chapter 2

Theoretical Foundation

I dealt only with visualisaion of measured data (representing topology of graph and its behaviour). Therefore, from the purposes of this work, a deep understanding to artificial neural networks was not necessary. This part provides only familiarization with studied problem for better understanding motivation to use these tools, procedures and ideas in order of creating each part and tool of application.

2.1 Artificial Neural Networks

Artifical neural networks are computational model of biological neural networks represented for example by human brain. Main advantage of neural networks is posibility of pararell programing. Therefore, these networks ara able to processing wide array of data in one moment. Also, the ability to generalize and associate gives the ANNs power to work with with uncomplete or noisy data.

ANN is compounded of basic computational fragments called neurons. Model of single neuron can be represented in couple of ways. The neural networkss of 2nd generation are widely used in these days. Compared to this, the 3rd generation ANNs are more biologically plausible and communicate with spikes. In this model, neuron is represented as a leaky integrator with threshold which integrates pulses on input multiplied by weight. Each pulse improves potencial of neuron and if threshold is exceeded neuron come excited and send pulse to other synapsed neurons.

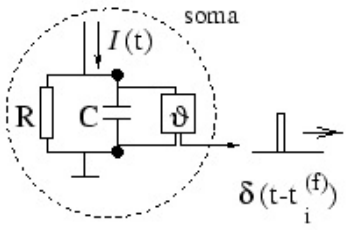

Figure 2.1: ***Leaky Integrator model*** *- Leaky integral-and-fire model of inner function of neuron 3rd generation (*Gerstner and Kistler*, 2012)*

Third generation ANN (also called SNN) is more biologically plausible than the 2nd generation. Properties of ANN are described in document (Example Depot, 2012). Signal is going through networks with delay (signal propagation has finite speed). Potential of neuron decreases with time when is not receiving a signal (pulses). Leaky integrator integrates incomming pulses (or input current) on inputs (dendrites). The integrated value decays with time. If the integrated value (so called inner potential) exceeds the threshold value, the neuron fires, that is: sends one spike to its output (axon). See 2.1.

Networks studied in my work do not have neurons on inputs. Inputs (input "neurons") have the shape of neuron but work only as distributors of the input signal.

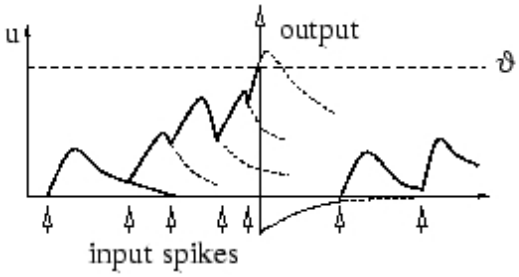

Figure 2.2: *Response of inner potential of neuron to incomming spikes. Neuron fires after receiving fifth spike on input.*

For purposes of simulations of these neural networks, there is tradeoff between biological plausibility and computation cost. We are using Izhikevich Simple model of neuron which is not so computationally demanding but should a lot of biologically plausible. The model is composed of two differential equations with four parameters (in our representation abcd). The parameter changes the behaviour and characteristics of neuron. Complete description is provided by (Eugene M. Izhikevich, 2003).

Third generation networks are computationally more demanding but they are able to produce more complex behaviour. It was shown that the second generation networks are only subset of the third generation.

The behaviour of these networks is often presented in a spiketrain. It represents information about spike occurance in time for each neuron. Each single time series provides information about times when the neuron became excited. More detailed description can be provided by (Maass, W., 1996).

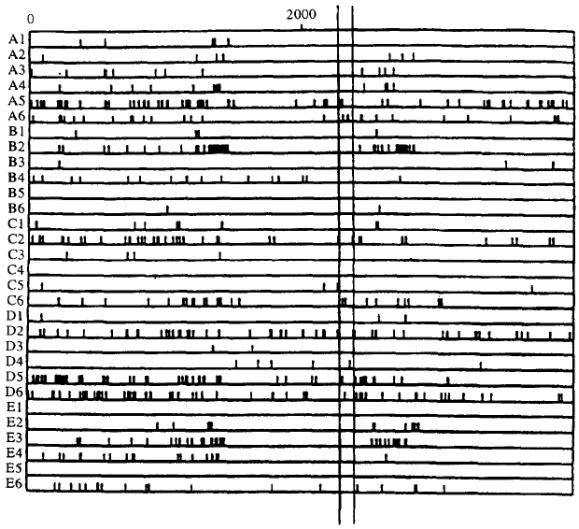

Figure 2.3: *Representation of spiketrain by Wolfgang Maass*

2.2 Graph Theory

In my work, I am working with reccurent neural networks. These networks I visualize as a graphs in 2D space. The graphs are formed by set of nodes and set of edges (Wikipedia, 2012). Neurons in the network are represented as nodes and connections between them as edges in our graph. In my work, I call this graph "topology". It means information about distribution of nodes in area and connection between nodes.

Developed and studied networks are formed of oriented edges. Each edge have two parameters: weight and delay which provides information about importance of edge in network (only weight, algorithms are not working with delay). Absence of some rules creates any specific kind of edges less common in standard graph representations. Bind can

connect one neuron itself. This situation have special representation in my application. Also, more than one edge can connect two neurons. This case is called multigraph in a graph theory.

Chapter 3

State of the Art

In this chapter I try to describe and discuss suitability for our purposes of available libraries designed to visualize and analyze ANNs or drawing logical graphs.

3.1 Source library

I wanted to build a visualisation tool, potentially based on some existing library. My goal is to create and test an alternative methods how to visualize and analyze the ANN topologies and behavior. We were looking for similar tool but we did not find any suitable. In that case, department decided to create a new instrument from the scratch.

3.1.1 Requiements

Me and my consultant Mr. Vítků have formulated few requiements of this Visualisation Tool.

- capability to display potentially huge networks
- zooming capability
- minimal computational complexity
- ability to draw the network topology (place the neurons into a 2D plane) according to some selected criterion
- ability to replay recorded behaviour of given ANN from the spiketrain

These were the basic criterions I used when studying and choosing library.

3.1.2 Options

Here is an overview of visualization tools found by me, which at least partially meet the requirements. In the following section I will summarize their main characteristics, I say which I finally chose the library and why.

1. (Piccolo2D, 2008*a*) - OSL supports drawing 2D graphs
2. (Neuroph, 2012) - OSL supports simulation of neural networks
3. (Guess, 2012) - OSL based on piccolo2D library with graph analysis methods.
4. (yWorks, 2012) - OSL supports good graphic representation.
5. (NeuroML, 2012) - OSL supports simulation of neural networks.
6. (Gephi, 2012) - OSL greatly supports analysis of graphs.
7. (Graphviz, 2012)- OSL supports drawing of oriented algorithm graphs.
8. (FANN, 2012) - OSL in C supports studying of neural networks.
9. (Encog, 2012) - OSL supports studying and learning neural networks.

3.1.3 Conclusion

Many libraries are adapted for simulation and development of neural networks. This type of libraries, such as libraries no.2,5,8 and 9, do not meet my requirements. Library no.3 and 6 were particularly interesting. They (3,6) contain methods and tools for drawing, analysis and visualisation of graphs. However, after I examined these libraries closer I found that no.3 uses methods of Piccolo2D library together with methods written in Python and no.6 is almost finished tool with little change. Libraries no.4,7 were unusable because they were primarily focused on better display of algorithms.

I chose the piccolo2D no.1. P2D is library developed on (University of Maryland, 2008). It includes tools for drawing basic objects and spatial manipulation with them. Furthermore, this library meets the most of my requirements and fully supports my style of work. I prefer to solve more complex problems using simple tools, methods and objects.

Chapter 4

Application Design

This chapter contains description of my application. The chapter provides inforamtion and details about object structure, tools for user, ranges of used variables and ways how my application draws and animates studied data. Also designed methods useful for better running of application are described.

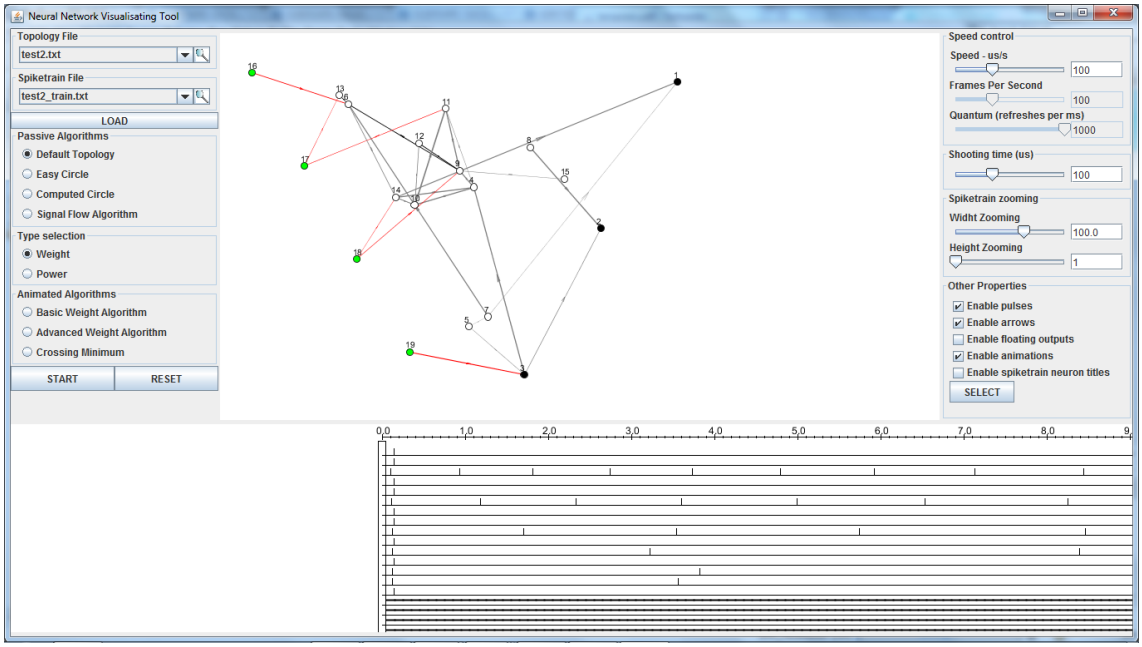

Figure 4.1: ***Application Appearance*** *- appearance of the visualizationtool displaying simple testing network.*

Application can load the network topology from txt file in a predefined format 4.2.1 4.2.2, after that it loads spiketrain data from another file. The application then draws both, the topology and the spiketrain. The application visualizes how signal (pulses) is

going through network in time. Also the user can change many properties of application, I designed and implemented, from speed and scales to enabling/disabling of many functions. To study network is prepared several algorithms which move neurons in space according to some selected criterion. The application does not do any simulations. It is used only for visualisating simulated data which are recorded in the spike train file.

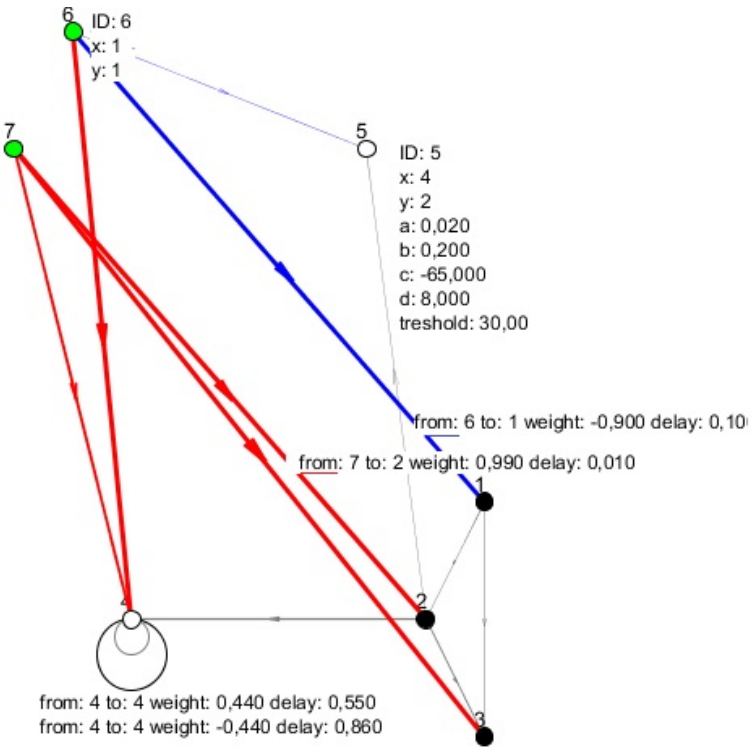

Figure 4.2: ***Network Basics*** *- Green neurons represent input neurons, black ones are output neurons and white ones are internal neurons. Input excitation binds are red and inhibition ones are blue. Other excitation binds are grey and inhibition ones are black. Width of bind is based on its weight. Input neurons do not have internal structure (abcd treshold) and work only as repeaters of input signal. The input signal can be analog (represented by current) or digital (represented by pulses).*

In this chapter will now follow descriptions of applications where a soon as possible about the description of the main objects, then provide examples for creating topologies, visualize the dynamic behavior of the site depending on the simulated data andthe possibilities of the users in the end .

4.1 Objects Structure

In this section I try to briefly describe object structure of application, each class variables and features in application.

- **Package Objects** - include basic structure of visualisated information.
  - **Bind** - Formal connection between neurons. Every bind has basic information about its start neuron (int from), end neuron (int to), delay on the bind in double and double weight.
    Less important is integer parameter. It is used for representation of multiplied binds between neurons.
  - **Neuron** - Formal interpretation of neuron. Class includes abcd and treshold parameters (each double), number of neuron (unique int ID), integer type of neuron (internal, output, input), analog/digital parameter (int AD) and list (ArrayList) of binds started in this neuron. Input neurons are realized only as repeaters and amplifiers of input signal. They do not have abcd parameters but they can repeat both analog and digital signal.
  - **Pulse** - Formal interpretation of pulse. The class determines both digital pulses and analog current. Every pulse has ID of shooting neuron (int from), ID of shot neuron (int to), delay (lifetime) of pulse in double, double time of pulse shoot, variable current (if it is different from 0 describes current) and bind which pulse is going trough.
  - **Spiketrain** - The class include lists (ArrayList) of currents and pulses. For better use, properties of visualisation class have variable for maximal current (maxCurrent) in double.
  - **Topology** - Main parameters of each topology is list of neurons (ArrayList), numbers of neuron types (int 1D field), abcd and theta parameters in (double 1D field). For better functions of the application is included boolean for enable floating outputs. More variables will be described in part algorithms.
- **Package Loading** Include part for loading spiketrain and topology from text files in predefined format 4.2.1 4.2.2.
  - **ReadSpikeTrain** - Loading of spiketrain
  - **ReadTopology** - Loading of topology

- **Package BNN Visualisation** The class is used for force first start and inicialize
- **Package GUIvisualisation** The class which is creating and managing GUI. Visualisation is divided into 3 single parts - network, spiketrain and GUI (options).
  - **Body** - Extended JFrame manages all accesiores.
  - **NetworkGUI** - The class extends PCanvas. All changes in network part are realised by this class. The class also has several methods for better reloading and automatic scaling.
  - **SpikeTrainGUI** - The class extends PCanvas and controls part of GUI where spiketrain is placed. It contains methods that enable user to change current time in paused mod by dragging the canvas.
- **Package ExternalControl** The class is created in order to have option to control the application by other applications. Control efforts are designed for maximal avoidance of collision.
- **Package edu.umd.cs.piccolo, edu.umd.cs.activities, edu.umd.cs.event, edu.umd.cs.nodes and edu.umd.cs.util** Packages are representations of imported library.

4.2 Data Loading

Application loads data from files ending with ".txt". Both documents have predefined format. This format is described in commentary of application parser_v0.3 created by my consultant Mr. Vítků. Also, the names of files have some rules that we accepted (me and Mr. Vítků). Corresponding topology file is named for example [name].txt and spiketrain file is named [name]_train[random code].txt. Where [name] is being code name of topology and random code is random sequence of characters for needs of more spiketrains for same topology.

4.2.1 Loading the topology data

This section is realized by class ReadTopology from package Loading.

Original formulation is described in parser v_0.3 (Vítků, J., 2012*a*).

Cited data format:

- parameters for Izhikevich neurons: [a b c d theta]
- number of input neurons
- number of output neurons
- total number of neurons (that is without the input ones)
- output neurons: [x,y] [a b c d theta] [to weight delay] ..
- other (hidden) neurons: [x,y] [to weight delay] ...
- input neurons: [x,y] [to weight delay] ..

where:

- x,y - are some coordinates of the neuron in the 2D plane
- to - ID (number of neuron numbered from 1)
- weigt - weight of the synapse
- delay - delay of the synapse
- abcd treshold - pars. are for hidden neurons specified globally

Application gathers information line by line. After each line Object Neuron is created and added to ArrayList of all neurons. Example on Fig. B.2.

Special situations can be solved here. When neuron is shooting himself bind's parameter shows linear increase. ANNs developed on department do not have any rules that would prevent them to create multiplied selfbinds. This parameter is useful in drawing multiplied selfbinds. Example in Fig 4.2.

4.2.2 Loading the spike train data

This section is realized by class ReadSpikeTrain from package Loading.

Original formulation is described in parser v_0.3 (Vítků, J., 2012*b*).

Citated data format (text reworked):

- number of input neurons

- number of output neurons
- total number of neurons (that is without the input ones)
- t1 t2 t3 t4 ...t55000 (list of spike times for neurons 1-total num of neurons)
- resolution
- bn efghijklmnopq

where:

- tn is time when neuron shooted
- resolution is time constant of current (time interval of one current value)
- bn is value which indicate if neurons is analog or digital
- efghij... are times of pulses for digital inputs nad times of current for analog inputs

Example on Fig. B.3

Application gathers information line by line. For each readed pulse time, application creates n pulses, where n is the number of neuron's binds. Each neuron has list of binds where it shoots on. Neuron does not know by whom is he shot.

Also special situations are solved here. If shooting neuron receives an analog input, pulse is created only for first bind and time calculation is linear based on parameter. If shooting neuron does not have receiver, pulse is created as selfpointing but with zero delay.

At the end of algorithm, list of all pulses is sorted by time and divided between list of currents and list of pulses. Sorting is useful for time managment of visualisation which will be described in the following text.

4.3 Topology Graphics

Application tries to draw network in a standardized format. It tries to display all neurons and make them similar for wide range of original positions and numbers of neurons. This is achieved by resizing positions of neurons and automatic scaling. User also gets good information about network parts (binds and neurons) after entering or clicking them.

Each neuron knows his position in two dimensional space. These positions can be saved in file topology of which is loaded from or calculated by any algorithm. These algorithms are described in chapter 5.

4.3.1 Drawing

Graphics is implemented in class networkGUI which extends PCanvas (inbuild class in P2D). Drawed network is divided into different layers. The layers are useful when creating listeners for huge amount of same components. The layers are: neuronLayer, edgeLayer, pulseLayer, identityLayer and analogEdgeLayer. Complete drawing and loading is realized by method reload(). It cleans the layers and draws new network.

Neurons in a network are painted by method drawNeurons(). This method places neurons on their default or calculated positions multiplied by netParameter variable. NetParameter is approximated by method calculateApproximteNetParameter(). The method calculates netParameter by setting average lenght of bind to 200 points. I chose this constant experimentally. I set color of neurons as green inputs, black outputs and white internals. Each neuron shows its informations after a click.

Binds are painted by drawEdges() and are divided between layer for input binds and layer for rest of binds. Other binds are all digital and their color is set to black for inhibition binds and grey for excitation binds. Input binds (binds from inputs) are colored red for excitation bind and blue for inhibition bind. If arrows are enabled, arrows, representing the direction of oriented edges (binds), are painted. Each bind has width based on its weight. Width is also normalized as division by weight of the bind with the biggest weight. This normalization makes each displayed network with same range of width $< -1,1 >$. Each entered bind shows its informations. If more binds overlaid on one another, application searches through all binds and information panel will contains informations about every bind between those neurons.

Drawing binds and neurons is inspired by code (Piccolo2D, 2008*b*).

Pulses are graphic representation of signal flow in network between neurons. All pulses are created as a red circles with height 2 and widht 2. They are made invisible. Library has problems with dynamic creation and removal of its particles so all pulses are created in every moment. For better numerical complexity, special access is used.

For each neuron there is an indicator which contains its ID. Information from graph are set to the same size in any zoom. Maximal unzoom is calculated in method CalculateMinimumScale(). This method finds neurons located on the extremes of x and y axis

and calculates zoom to show the whole graph. Library has default zooming tools. They are disabled and zooming method is created for mouseWheel manually. Each rotation difference changes scale by 10%. Analog inputs have second indicator which dynamicly changes its value in order of current flowing from the input.

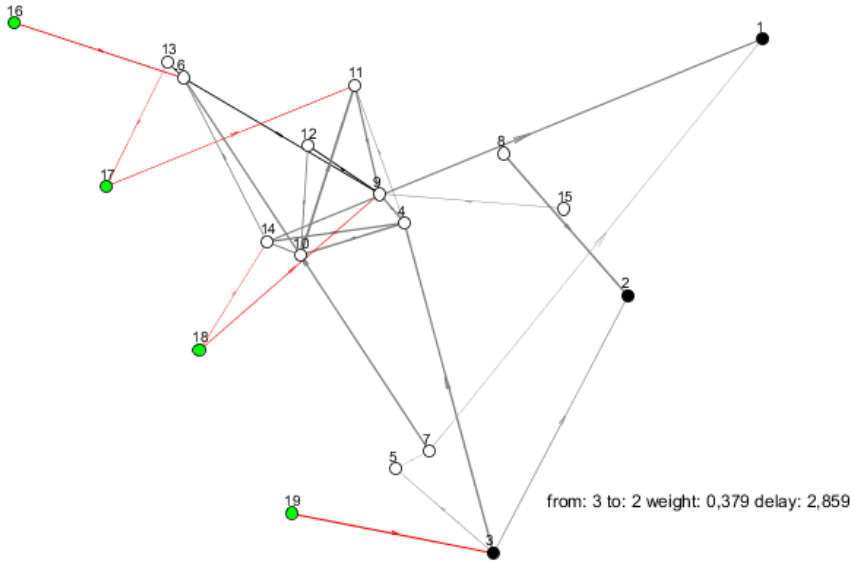

Figure 4.3: ***Testing Topology*** *- example of simpler topology drawed by application. Also is showed how application provides information about binds.*

4.3.2 Dynamics

This section provides introduction to displaying internal dynamics of the network. The refreshing process will first summarized and then each part briefly described.

Application contains few methods which are useful for showing internal states of the network. These methods change the excitations of neurons, flow of signal in network and power of current going through the network. Inputs are renewed, depending on the input signal. For current on inputs, analog refreshing method is used. In other case, input signal is administered with other pulses in network. Also, tools that show the information about parts of the network are managed here.

Refreshing is realized by methods refreshCurrent(double time), refreshExcitedNeurons(double time) and refreshPulses(double time, boolean full). Work time of application is sized in miliseconds. Resolution of time is based on variable quantum. Quantum is

sized in range from 1000 pulses per milisecond to 0.001 pulse per milisecond. This wide range allow application to animate in resolution from microseconds per second to seconds per second.

If "enable network pulses" is allowed this method refreshes pulses(double time, boolean full), changes positions of displayed pulses and reveals newly shot pulses. This method is searching the list of pulses (sorted by time) from down constraint (the oldest pulse) to first unshooted. When the oldest pulse reaches shot neuron, new the oldest pulse is found. If boolean full match true, all of the pulses are refreshed. It is useful when time is changed from different place in program, for example by dragging the spiketrain.

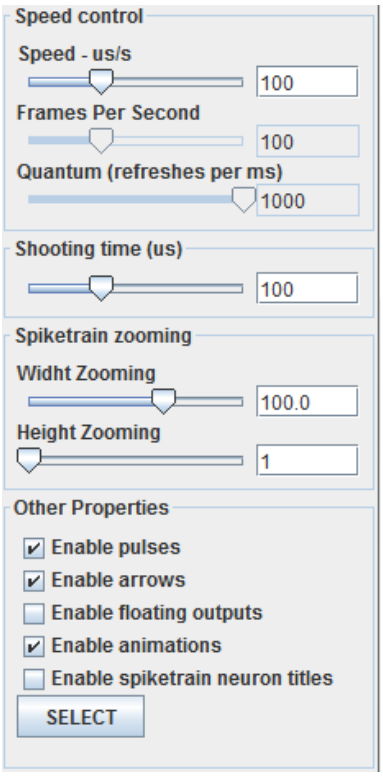

Figure 4.4: ***Application Tools*** *- right panels which contains all non-loading and non-algorithm (choose) tools.*

Method refreshExcitedNeurons(double time) searches pulses that are shot at half of shootTime(lenght of exciting neuron) before or after the real shoot time of pulse. If neuron is excited, it changes its color to red.

Method refreshCurrent(double time) works same as refreshPulses(double time). It uses a list of the currents and downAConstrain. When method finds curent in the right time, it changes the value of analog indicator near the neuron and changes color of binds.

Color is mixed from RGB and function is:

Exhibit

$$RGB = (255, 100 - 100 \times \frac{cur}{maxCur}, 255 - 255 \times \frac{cur}{maxCur}) \tag{4.1}$$

Inhibit

$$RGB = (255 - 255 \times \frac{cur}{maxCur}, 100 - 100 \times \frac{cur}{maxCur}, 255) \tag{4.2}$$

where cur is value of the current and $maxCur$ is maximal value of the current that is founded from list of currents. For negative values of current functions are opposite. Green is mixed for better contrast of the zero current.

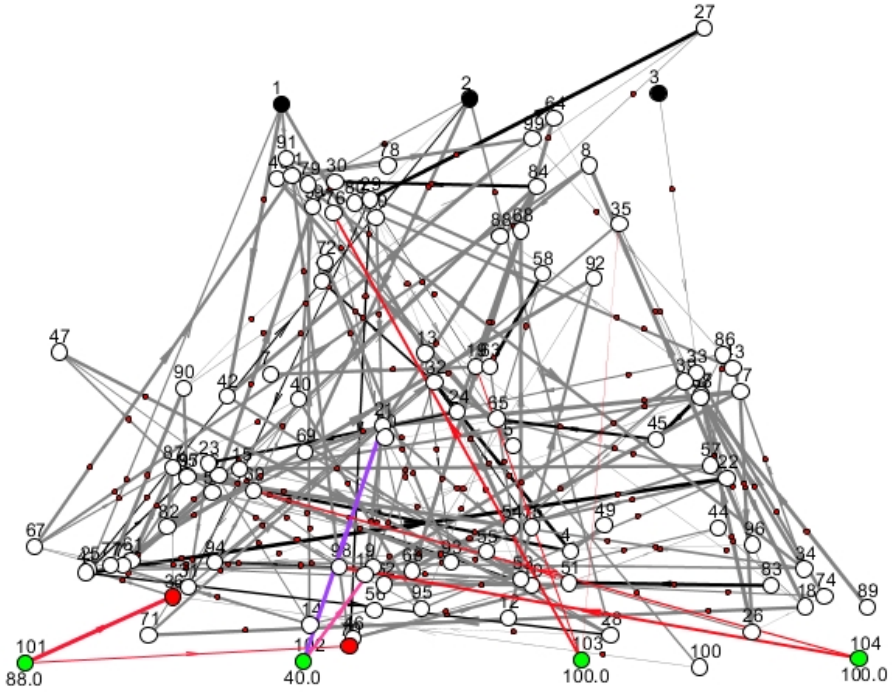

Figure 4.5: ***Dynamics*** *- state of network in animation. Pulses are travelling through the network, neurons marked red send pulses at the current time step and analog values and color are changing.*

4.4 Spiketrain Graphics

The application can draw spiketrain. The spiketrain appears when pulses were sent from all neurons (one for each line). the spiketrain has a timeline showing the current position

in the simulation time. Additionally, the application displays timewindow, which is used for two main things at once:

It can set how long neuron in topology to see it shot up

It can set the parameters of algorithms that work with dynamic behavior site, see below Chapter 5.

Application allows to display spiketrain zoomed in wide range freely in both ways (heigh and width). Also, user can pause the simulation and change time manually with corresponding changes in network. Application visualises function of ANN in real time. User can set speed of animation from microsecond per second to real time (second per second). Position of spiketrain and time line corresponds with position of pulse in network.

4.4.1 Drawing

Spiketrain graphics is handled in class SpiketrainGUI which extends PCanvas. Spiketrain is divided between pulseLayer that carries all the pulses and neuronLayer that carries all the remaining components. Pulses are represented by vertical lines that are set on positions based on their shoot time and shooting neuron. Analog input current on input neurons is represented by horizontal lines. Positions are calculated same as vertical ones but their lenght is based on parameter of spiketrain.

SpiketrainGUI has for drawing variables magnificationX and magnificationY. These parameters are changed by user and their function is zooming independently in y or x axis. X axis is scaled for $magnificationX = 1$ as ms per second.

Spiketrain is drawed by method drawSpikeTrain(). First, timeWindow and timeLine are created. Time line on moving spiketrain represents current time. It is useful for higher width of shootingTime. TimeWindow is rectangle whose width is calculated by equation:

$$width = shootTime \times magnificationX \tag{4.3}$$

Height of canvas for spiketrain is sized to 250 pixels. Height added to every neuron of topology is calculated by equation:

$$heightConstant = \frac{250 \times magnificationY}{numberOfNeurons + 1} \tag{4.4}$$

This height is resized to integer (number of pixels cannot be divided) and its value is restricted to minimum of 2 pixel for single neuron. For each neuron there is a drawed

line in the lenght of time of latest pulse plus 2 miliseconds per heightConstant.

For each pulse there is created a vertical line high $\frac{2}{3} \times heightConstant$. For each current fragment there is created a horizontal line in height of the current and width of the spiketrainParameter multiplied by magnificationX.

Method also draws time axis. It is drawed for resolutions of 0.1 miliseconds for magnificationX bigger than 30 and 0.1 seconds for magnificationX smaller than 30.

For enabled neuron titles there is created PText (text lable suitable for PCanvas) with names of all neurons and heigh constant is resized for correct displaying of neuron IDs.

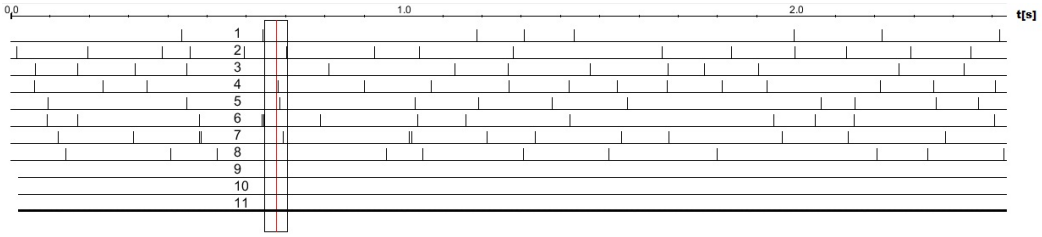

Figure 4.6: ***Testing Topology Spiketrain*** *- Spiketrain with time line and time window. We can see time window contains four pulses - neurons 2,4,5 and 7 will be displayed with red color (these neurons just fired). Most neuron contains spikes of digital neurons and bottom line represent analog signal on input neuron 11.*

4.4.2 Dynamics

Spiketrain refreshing is realized by method refreshSpiketrain. Method changes position of timeWindow, timeLine and neuronsNames depending on the time. Spiketrain is displayed in GUI by PCamera (part of P2D). Method also changes displayed field and if neccesary, changes the position of neuron titles. Neuron titles are fixed in position less than 200 pixels far from the time line.

If application is paused canvas can be dragged. Canvas cannot be dragged out of the bounds. Action of dragging starts method passiveRefresh which calculates current time of displayed window and starts time refreshing sequence of application. It means that when canvas is being dragged network dynamicly changes according to the calculated time. For this kind of time change there is a method from networkGUI - refreshPulses() suited for complete refreshing of all pulses. It avoids any problems with inteligent searching in list of pulses.

4.5 Graphical User Interface

In this section I briefly learn features designed for user interaction with the application. I will describe how I decided to split all the features and each will explain shortly

GUI allows user to change lot of application properties. User can easily control everything he needed to use. All these fuctions are clearly placed in three sections divided between two panels. Left menu contains file loading and command buttons for control algorithms. Right menu contains controls for changing properties of drawing and running of animations. All functions were tested to attain best comfort and stability of application possible.

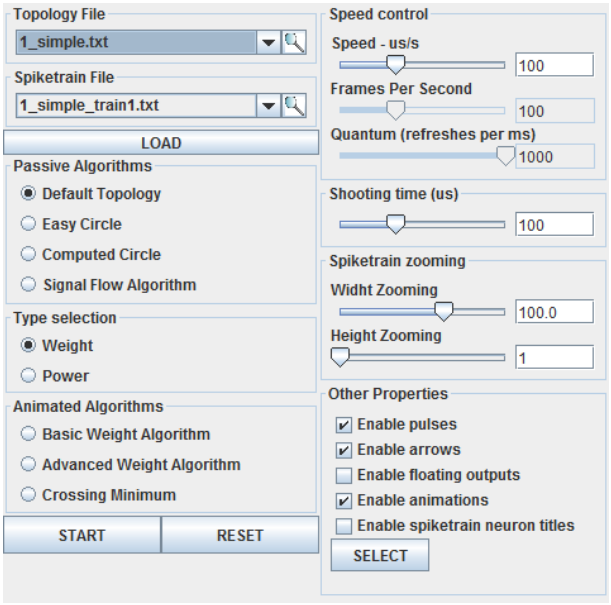

Figure 4.7: ***Graphical User Interface*** *- left panel contains loading part and part for selection and setting up the parameters of particular algorithms. Right part contains tools for operations with application parameters and button for selective searching (or possible X-ray).*

4.5.1 File loading

Selection of topology file and spiketrain file is realized by JComboBox. I formulated the location of spiketrain files to the directory called train in project directory and topologies saved in directory topology. Topology files cannot contain the sequence "train". Incorrectly saved files will be ignored or will not get through the second control sequence. It

is done as an easy string control of names format. Spiketrain and topology names must end with .txt and spiketrain name must contain everything except .txt of topology name.

User also has oppourtunity to browse his own computer for directory containing possibly useful files. For better orientation input mask is set compatible with file format. When browsing of other directories is finished, user can easily right click on lens button which selects the default directory back again. This tool is also useful for refreshing contents of boxes.

Directory search is realized by method directorySearching(String topology, String spiketrain). This method searches directory topology and finds all the files ended with ".txt" and in directory train finds all suitable files and fills them into JComboBox for spiketrains. Method is inspired by code (Example Depot, 2012).

Load button starts the load sequence. This sequence is realized by method loadTopologySpiketrain() which load topology and spiketrains from selected files. Boot(boolean full) is called afterwards. Function redraws network and spiketrain. Boolean full is useful for drawing the topology only through class external control (when spiketrain is not necessary).

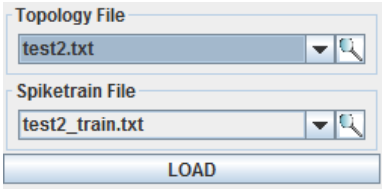

Figure 4.8: ***GUI for loading files*** *- Offers of files saved in current directory, different than defaul directory may be opened by lens button. By right click default directory will be searched.*

4.5.2 Tools menu

The application offers to user wide area of interaction. This provides great freedom and control over aplication. Each function will be shortly described and effects of changes will be explained. All tools are placed in the right menu. Right menu is created by method createRightMenu() and objects are controled from refreshTools().

1. **Speed Control** For user there is possible to set speed of simulation vizualization. It means speed of animation. Because animation has nonzero calculation time

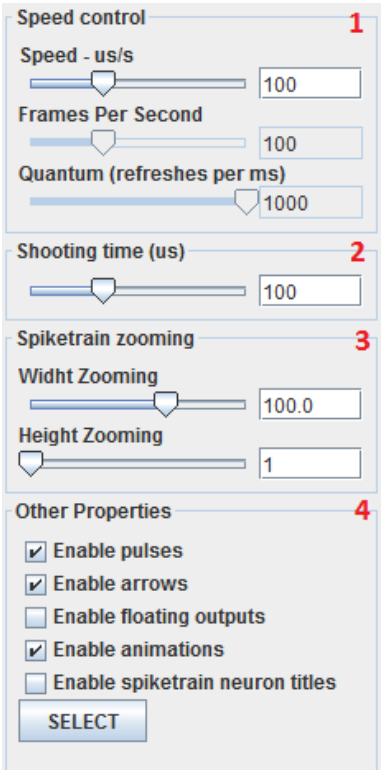

Figure 4.9: ***GUI for operation with animation and drawing*** *- speed sets real speed of animation (microseconds per second), Frames per second display computational speed of application (running visualisation or animated algorithm), currently automatically set quantum, lenght of excitation time, width and height zooming of spiketrain, other enabling attributes and button for selective choosing or possibly X-ray use*

per one refresh automatic correction is requested. This correction is realized by refreshSpeed(). This method calculates predicted time of one cycle. If predicted time for current quantum is shorter, quantum is decreased. If predicted time is ten times longer quantum is increased. From predicted delay and real cycle time is calculated additional delay which adjusts full delay to be as big as predicted.

Frame speed and quantum show current values to the user. Frame speed indicates number of cycles calculated by computer in current speed. Speed ranges from 1 microsecond of animation per real second to 1 second of animation per real second. Quantum range is sized from 1 refresh (cycle) per second (0.001 of refresh per milisecond) to 1.000 refreshes per milisecond (1.000.000 refreshes per second). Thanks to this application, time control is suitable for long range of computers and networks.

2. **Shooting Time** Shooting time affects how long period of time the application indicates that the neuron sent spike to its output and based on this parameter is calculated power bind by method createListOfPowerBinds(). See in Section 5.7. Range is sized from 1 microsecond to 1 second. Change this parameter immediately alters the width of the time window.

3. **Spiketrain zooming** Position of pulses in spiketrain is calculated as a shooting time (in miliseconds) multiplied by width zooming (in x axis). Range is sized from 0.001(1 pixel per second) to 100.000 (100.000 pixels per milisecond). Height zooming is paramter which multiplies calculated height constant. Range is scaled from 1 to 15. If enableNeuronTitles is changed, range is rescaled according to the height constant.

4. **Other Properties** This part is useful for enabling several of application functions. "Enable pulses" button affects visualisating of pulses. For enabled pulses, method refreshNeurons(...) is called. "Enable arrows" affects drawing of the arrows useful for orientation of binds. "Enable floating outputs" is useful for algorithms. Enabling allows algorithms to change positions of outputs. "Enable animations" adds or removes short delay between two cycles in animated algorithms (useful for too small networks or too powerful computers). For normal (for example Dual core 2.2GHz) or slower this opportunity changes are not needed because algorithms will be animated in both ways. "Enable neuron titles" shows titles for better orientation in spiketrain. Also, height zooming can be recalculated.

4.5.3 Algorithms menu

Algorithms place neurons in 2D space by specific criterion. They work with actual positions of neurons. Results directly depend on the order of running algorithms.

They can work with passive or dynamic kind of binds. Passive are real binds in network and dynamic are based on simulated data from spiketrain (this reflects the behaviour of network in simulation).

This part of GUI allows user to apply algorithm on topology in its current state. Each algorithm places or moves neurons by specified criteria. Each single algorithm is described in chapter 5. Some criterions are based on properties set in section tools menu.

Menu is divided in three parts:

- One part offers the selection of which kind of binds we want to study. Weight (passive) is opinion where application calculates with binds between neurons based on real binds between them (binds from topology). Power (active) opportunity creates new list of all binds based on probability of exciting. Bind is more stronger, one neuron shoot after the other more often. More details are described in algorithms part.

- Second part contains passive algorithms which are not calculated step by step. This part includes default distribution (based on positions in file). These algorithms are generally computationally less demanding than animated.

- Third part contains active algorithms which are calculated steb by step. For animated algorithms method repaintNetwork()is best suited. This method changes positions of all current objects in network. It is safer than repetitive reloading. User have also choice to stop animated algorithm from his own will by using stop button. Action perform interuption of algorithm and network stay in state after last refresh.

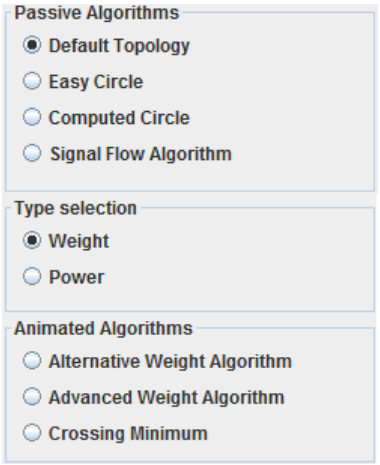

Figure 4.10: *GUI for choosing algorithms*

Now I will try to briefly describe all the algorithms and their functions. Complete information is contained in Chapter 5.

- **Default topology** places neurons to default distribution loaded from file. Positions are saved in two dimensional field and loaded from it in class Topology.

- **Easy circle** places neurons on circle in dependence of their ID, used for placing inputs and outputs to stable position and for comparison with computed circle.
- **Computed Circle** places input and output neuron to stable positions and calculate possibly best distribution of neurons on circle in order to improve clarity.
- **Signal Flow algorithm** places input neurons to stable positions and expands neurons to layers with emphasis to minimal crossing.
- **Weight** creates list compound of real binds between neurons.
- **Power** creates list of binds based on shooting probability. This bind is heavier between two neurons more often is one excited after second in time range based on shooting time.
- **Alternative Weight Algorithm** moves with neurons in direction of optimal lenghts of binds based on their weights.
- **Advanced Weight Algorithm** tries to find optimal positions for neurons based on bind weights and distances between neurons.
- **Crossing Minimum** is trying to change neuron positions to make complete crossing of network minimal.

Chapter 5

Algorithms and Experiments

In this chapter I will introduce chosen algorithms designed for assignment of neurons in two dimensional space. Algorithms are designed to reflect different criterion, which are potentially useful for better understanding the network structure or behaviour.

I willl describe each algorithm's principle, what would tell us about the structure or behavior of a network, discuss its potential advantages and disadvantages.

For all algorithms the following common: the input network topology (eventually spiketrain data), containing both neuron positions, some criterion algorithm into account, and the output are new positions of neurons on the 2D space. Input and output neurons may or may not have to be fixed on their position.

5.1 Easy Circle

Easy circle is the most basic algorithm I implemented in my application. First objective is to place output and input neurons to predefined positions. Algorithm places input and output neurons on predefined positions and internal neurons are drawed into circle. Internal neurons are placed on circle based on their ID. The algorithm can be used for comparison of results given by Computed Circle algorithm 5.2.

Easy circle is created by method createCircle(boolean easy). For possitive easy, neurons are arranged according to their ID, if not, they are placed according to the calculated result (int 1D field). Result contains calculated sequence of internal neurons.

New (static) coordinates of input neurons are computed according to the following

equations [1]:

$$x = 0 \tag{5.1}$$

$$y = 300 \times \frac{(ID - o - n - 1)}{i} \tag{5.2}$$

Where o is number of outputs, n is number of internals and i is rank of input neuron.

New (static) coordinates of output neurons are computed according to the following equations:

$$x = 300 \tag{5.3}$$

$$y = 300 \times \frac{ID - 1}{o} \tag{5.4}$$

Where o is number of outputs.

Finally, hidden neurons are placed as follows:

$$x = 150 - 100 \times \cos \frac{2\pi \times (ID - o)}{n} \tag{5.5}$$

$$y = 100 + 100 \times \sin \frac{2\pi \times (ID - o)}{n} \tag{5.6}$$

Where o is nuber of outputs, n is number of internal neurons and ID is unique identificator of given neuron.

5.2 Computed Circle

Algorithm tries to place neurons in better order for clearer results. The main idea of the algorithm is to place strongly (and/or many times) connected neurons closely together. Algorithm searches through the list of binds and makes connection between neurons when both bind neurons are not connected to two another neurons.

My algorithm often use representation of network in a data structure called by me SolveMatrix. It is a two dimensional antidiagonal (elements on diagonal are zero) adjacency matrix. The field contains information that two neurons are placed nearby. Each neuron can be binded, see Fig. 5.2 to maximum of two another neurons. This rule is based on a premise that each neuron on circle can have only two neiboughrs. Bind is represented by number 1 in matrix. It indicates presence of bind between these neurons. Zero presents absence of bind between these neurons

[1] Symbol $\times$ represents the standard multiplication sign (not a vector or otherwise) in whole text.

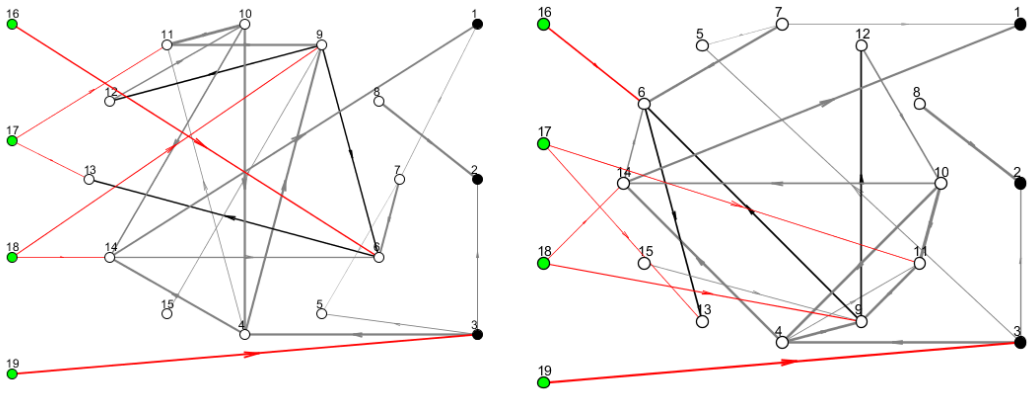

Figure 5.1: *I took easy topology with distribution of neurons in Fig. 4.3 and on figure we can see the results. Left network is for "easy circle" and right result is for "computed circle". We see that "computed circle" algorithm tries to place strong binded neurons nearby. In finished state, computed circle has less crossings in the midle of circle and is clearer.*

In its first part solveMatrix is filled from the strongest to the lightest bind in workBinds list. It means, algorithm gradually takes each bind and summarize solveMatrix row corresponding with shooting neuron and row corresponding with shot neuron. If both neurons have less than two neibourghs (are connected to one or zero neurons) ones on corresponding positions in matrix are set up.

Second part of algorithm solve outright on circle (saved to field result). If algorithm does not have sequence of neurons it finds first neuron with single binded neuron which is possibly pheriperal for different sequence.

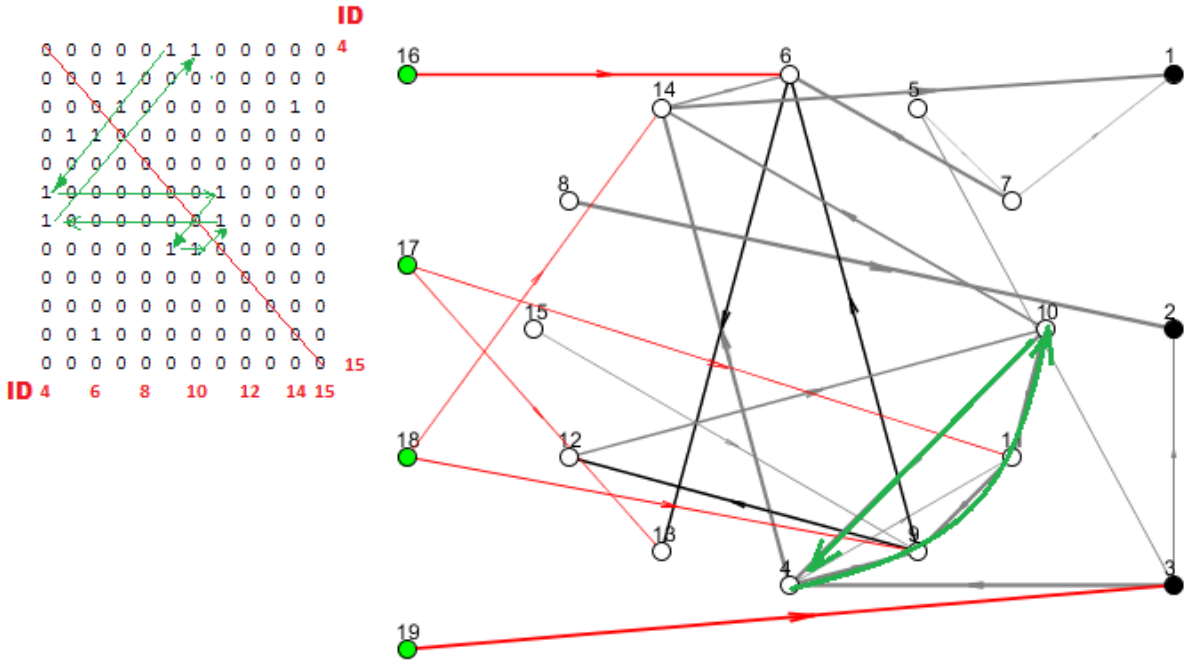

Figure 5.2: ***Solve Matrix*** *extracted for smaller topology. Diagonal elements have zero value. Ones in matrix represents the existence of bind between the neurons and zeros absence of bind. Green color shows the loop in solve matrix and its analogy at the final outcome.*

Data: workBinds
Output: result 1D field
for *each bind* **do**
  $n = \sum_{j=0}^{i}(solveMatrix[ID_t - 1 - o][i])$ `// o - outputs, i - internals`
  $m = \sum_{j=0}^{i}(solveMatrix[ID_r - 1 - o][i])$ `// n,m - numbers of binds`
  if n <2 `and` m <2 **then**
    $solveMatrix[ID_t - 1 - o][ID_r - 1 - o] = 1$ `// r - shot neuron`
    $solveMatrix[ID_r - 1 - o][ID_t - 1 - o] = 1$ `// t - shooting neuron`
  end
end
$line = 0$
for j = *1 to* i **do**
  $result[j] = line$ `// i is number of internals`
  $chosen[line + o] = true$
  $line = ID_r$ `//` ID_r `is number of column of next neuron`
  if *chosen[line] ==true* **then**
    for k=*1 to* i **do**
      $n = \sum_{l=0}^{i}(solveMatrix[k][l])$
      if n==*1* **then**
        $line = k$
        $k = i$
      end
    end
    if *single binded neuron not found* **then**
      while *chosen[line]==true* **do**
        $line = random(0, n)$
      end
    end
  end
end

Algorithm 1: **Computed Circle Algorithm** - Circle is computed as 1D field of neurons that are painted by method createCircle(boolean easy)

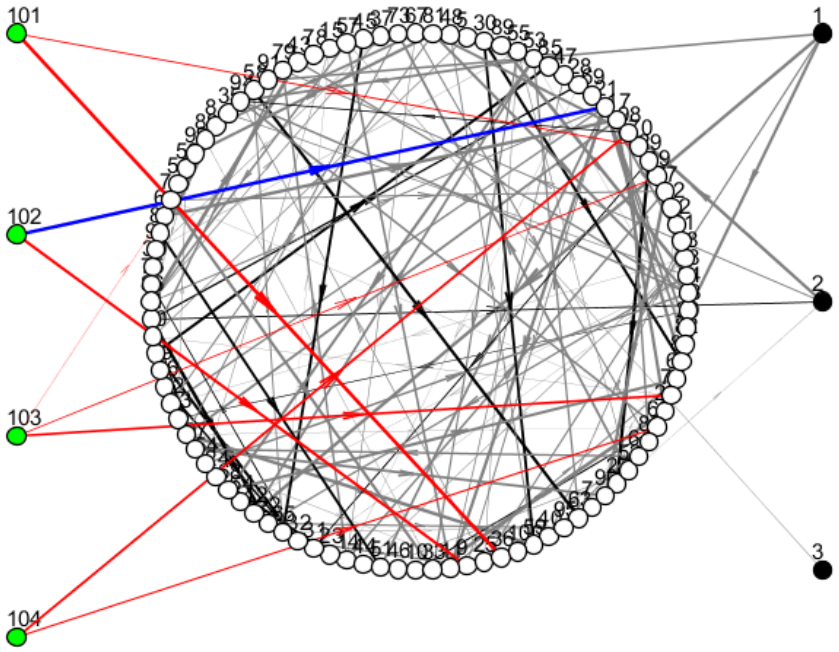

Figure 5.3: *For larger networks results provided by computed circle algorithm became practically unreadable for human. In order to display bigger networks in some readable format, I designed and implemented another algorithms, described below.*

5.3 Basic Weight Algorithm

This algorithm was proposed by me. The main idea is to place the neurons into the 2D space in a way that stronger interconencted neurons will be closer to each other than those less interconnected ones. While this is NP complete task, I designed the following local optimization algorithm. Unlike Advanced Weight Algorithm it can move two neurons in one step and is strong enough not to fall to the local static minimum. Algorithm does not respect any other criteria, so the different positions of non-floating outputs and inputs can lead to non-optimal and confused results.

In each step algorithm iterates through list of all binds workBinds() which are sorted by weight. For every bind it calculates ideal positions on line (that connect both neurons) and move them one small step closer to the ideal position. This is happening until changes in one cycle are different than in the previous one. Step is calculated in class Topology calculateStepBasicWeightAlgorithm().

Data: workBinds, lastDistance
Result: new positions
for *each bind* **do**
  $distance = \sqrt{(x2-x1)^2 + (y2-y1)^2}$ `// x1,y1 shooting neuron positions`
  $totalDistance+ = distance$ `// x2,y2 shot neuron positions`
  $prefferedDistance = (1 - bind.weight) \times 100$
  $change = \frac{-prefferedDistance}{distance} + 1$
  if *floating outputs enabled or neuron = internal and neuron != input* **then**
    $x1 = x1 + (x2 - x1) \times change \times bind.weight$
    $y1 = y1 + (y2 - y1) \times change \times bind.weight$
  end
  if *floating outputs enabled or neuron = internal and neuron != input* **then**
    $x2 = x2 + (x1 - x2) \times change \times bind.weight$
    $y2 = y2 + (y1 - y2) \times change \times bind.weight$
  end
end
if $totalDistance < lastDistance$ **then**
  return true
else
  return false
end

Algorithm 2: **Step of Basic Weight Alhorithm** - calculate small steps to locally optimal positions in weights criterion of binds

5.4 Advanced Weight Algorithm

Algorithm's architecture is based on finding better vital function in surroundings of a neuron. Main criterion depends on bind weight. Vitality is evaluated how much is the current state of the neuron chart correctly (similar to a fitness function). Vitality is calculated by following equation:

$$vital = \sum_{n=0}^{b}(d - (1 - w) \times 100) + \sum_{n=0}^{c}\frac{10}{d} \tag{5.7}$$

where b is number of neuron binds, d is distance between binded neurons, w is weight of bind and c is number of neurons. Algorithm is trying to set neurons to the best possible distances that depend on the bind weight and, less importantly, on attempt to maximize distances between neurons.

I suggested this algorithm because the results of "basic weight algorithm" were often confused and "basic weight algorithm" can not accept more criterions for fixing it. In addition, "basic weight algorithm worked" with ties gradually and not simultaneously with the whole state of topology. Due to differences in approach, I can run "advanced weight algorithm" with the state of the entire network and work with other criterions useful for better visibility of network.

```
Data: topology
Result: new positions
for each neuron do
    minVital = getVital(neuron)
    minPosition = currentPosition
    for surrounding positons do
        set neuron on surrounding position
        vital = getVital(neuron)
        if vital <minVital then
            minVital = vital
            minPosition = current surrounding position
        end
    end
    set neuron on minPosition
    if any neuron changed position then
        return true changed
    else
        return false changed
    end
end
```

Algorithm 3: **Step of Advanced Algorithm** - calculate on step of advanced algorithm and change change positions of neurons

5.5 Advanced Crossing Algorithm

Main criterion is here the minimization of crossing binds between neurons. This algorithm should draw the networks where the crossing is minimazed, therefore the resulting picture

should be more understable (clear) to the human.

The algorithm is based on finding better vital function in surroundings of a neuron. Main criterion depends on minimal crossing. Method for realization is optimizeCrossingsForNeuron(int ID) that changes matrix crossing only for rows and columns of bind connected to the optimized neuron. Class topology includes crossingMatrix(2Dimensional double) that brings information about crossings of every two binds. Crossing of two binds is calculated by method getCrossingValue(bind1, bind2). This method finds intersection of two lines described by parametrical equations. If all 4 parameters are in range from 0 to 1, intersection lands on both binds and binds are crossing.

Crossing value is calculated as:

$$crossingValue = bind1.weight \times bind2.weight \tag{5.8}$$

Vital function is calculated as:

$$vital = \sum_{n=0}^{b}(d - (1 - w) \times 100) + \sum_{k=0}^{i}\sum_{j=0}^{i} crossingMatrix[j][k] \tag{5.9}$$

where b is number of binds that connect neuron, i is number of internal neurons, w is weight of bind and d is distance between neurons.

Algorithm uses the structure of advanced algorithm. Function differs in size of the surrouding place (step) and in function of calculating vital function.

5.6 Signal Flow Algorithm

We are displaying generally recurrent neural netowrks, but even in some of these networks could be some main (global) direction of signal flow (from input neurons to output neurons). This algorithm basically tries to display the network in a similar way how are feedforward networks commonly represented. The neurons are placed in layers in direction from input layer towards the ouput neurons. For other choice ouput neuron can be placed in players like internal neurons. This choice can show how close possibly ouput neurons are to input neurons. The algorithm is inspired by the Fig.2 in the wiring diagram of C. Elegans in this document: (L.R. Varshney, B.L. Chen, E. Paniagua, D.H. Hall and D.B. Chklovskii, 2012). The algorithm tries to lay the neuron layers as is the case of normal distribution of neural networks. Result which I am trying to achieve is that most of the signal (pulses) went in right direction. I assume that the

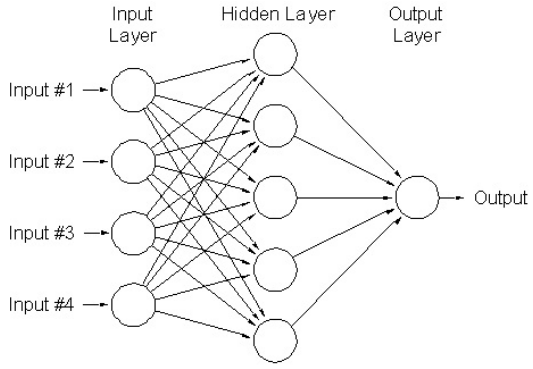

Figure 5.4: *Schematics of feed forward ANNs.*

signal will spread gradually from the input layer and only at a later stage in the network behavior will reflect the feedback binds.

Graph is created by layers. Neurons from new neuron are expanded in the next layer by rule of minimal crossing. Position on new Layer is found by the way that by step is calculated functions for n positions. In neibourghood of minimum local minimum is found with decreasing step.

Algorithm is handled by method calculateSignalFlow(). At start method places input neurons according to the following equations:

$$x = 0 \tag{5.10}$$

$$y = 50 \times i \tag{5.11}$$

Where i is rank of input neuron.

In each cycle neuron from open list is chosen and his unplaced children are placed and added into open list. If open list is empty and no neurons are placed first unchosen neuron is placed and added to open list. When negative, "enable floating outputs" method choose output neurons last.

Placing of neurons on layer is realized by method flowLandingAlgorithm(Neuron neuron, int layer, int parentY). Method tries to find minimum on vertical line which x position is:

$$x = layer \times 5 \tag{5.12}$$

Minimum is searched from Y position -50 to 200 pixels. Vital function is calculated

by method getVitalFlow(Neuron neuron, int parentY). Vital function is:

$$vital = \sum_{j=0}^{n} |0.1 \times (neuron.y - n(j).y)| \quad + \\ 0.1 \times |parentY - neuron.y| \quad + \quad \sum_{k=0}^{i} \sum_{j=0}^{i} crossingMatrix[j][k] \quad (5.13)$$

where n is number of neurons on same layer, $n(j).y$ is y position of neuron on same layer and i is number of internal neurons. Before summarization of crossingMatrix, crossingMatrix is changed by method optimizeCrossingForNeuron(int ID). Method recalculates crossing of binds of chosen (placed) neurons.

The first unchosen neuron has not included part with parent.

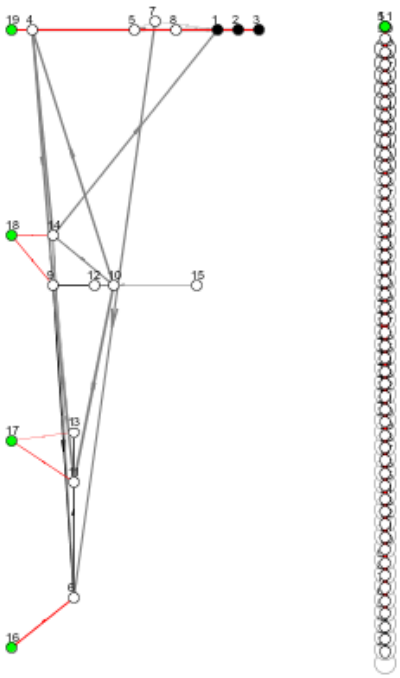

Figure 5.5: *Left (distribution on Fig. 4.3) topology shows that algorithm is not suited for small networks but the algorithm returns quite good results. Second (random 50) show that signal flow is useless fully connected reccurent neural networks, where the signal flow has no some specific direction.*

On Fig. 5.6 we see example of expected result. But this algorithm has also specific cases which can not be solved by this approach. On Fig. 5.5 we see on right example of fully connected network where signal has no specific direction.

5.7 Power

Radio button named Power creates list of power binds. These binds reflects how much together are neurons shooting. It means that neurons which are shooting often together probably will be sticked. This algorithm can be set up by the parameter shooting time. The shooting time can be set from GUI and is displayed by the timeWindow. The wider shooting time, the bigger time window is, and the more neurons are fire together.

Input: spiketrain, shootTime
Data: probabilityVector, probabilityMatrix
Result: workBinds list
for *each pulse* **do**
  make empty vector `// second neuron can shoot more than once`
  $time = pulse.time$ `// time of searched pulse`
  for *each pulse2 older than pulse and time* + $shootTime$>$pulse2.time$ **do**
    $probabilityVector[pulse2.from - 1] = 1$ `// pulse2 shot in interval`
  end
  while i<`number of neurons` **do**
    $probabilityMatrix[pulse.from][i] += probabilityVector[i]$
    if $i == pulse.from$ **then**
      $probabilityMatrix[i][i]++$ `// matrix[i][i] has number of shoots`
    end
    $i++$
  end
end
for *each row of probabilityMatrix* i **do**
  for *each column of probabilityMatrix* j **do**
    if *probabilityMatrix[i][j]*<0 **then**
      workBinds.add(from i+1, to j+1,delay 0,parameter 0,
      weight $\frac{probabilityMatrix[i][j]}{probabilityMatrix[i][i]}$)
    end
  end
end
sortByWeight()

Algorithm 4: **Creating of Power Binds** - process how power binds are created.

5.8 Experiments

Algorithms can show different regularities of neural network based on used algorithm and type of binds. All parameters used in experiments will be described. If any parameter will not be described his value was not changed from the starting value of the application.

5.8.1 Signal Flow on Moderate Network

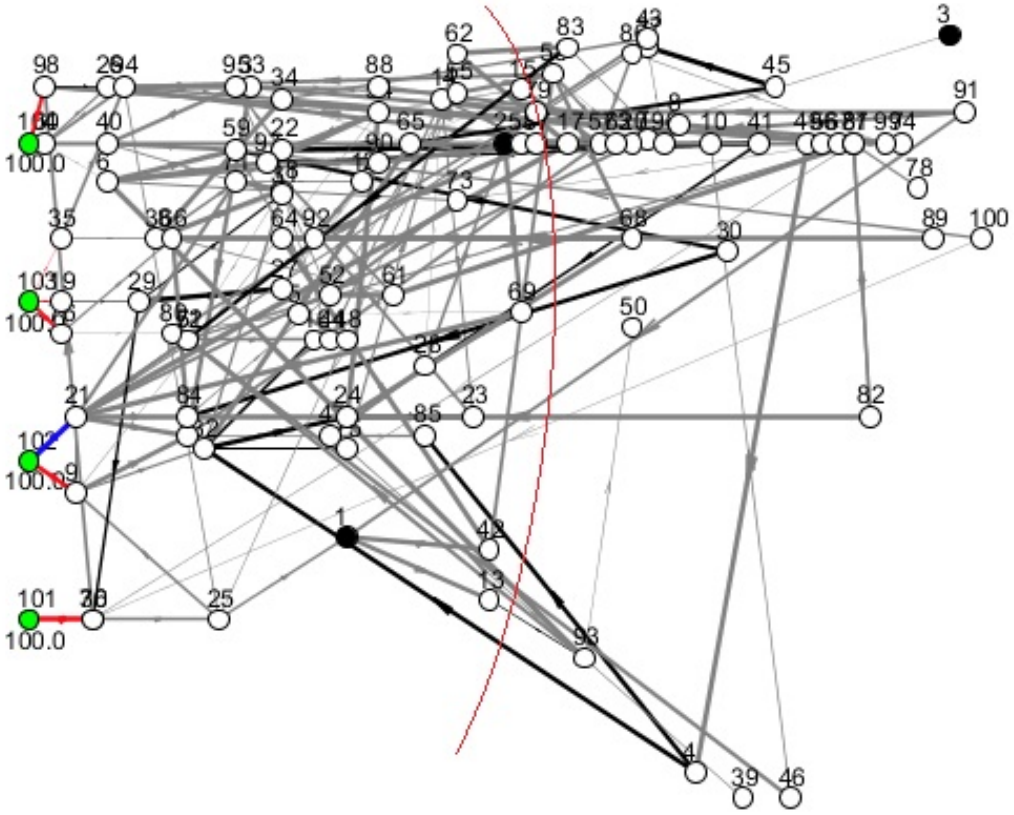

Figure 5.6: ***Signal Flow on Moderate Network*** - *Signal Flow algorithm was used to network on Fig. 4.5. The outputs were also permitted location anywhere in the graph.*

By this placing of neurons to a 2D space, we learn about the inner structure of the network. We can see which neurons are important and which are, however, most probably for nothing.

The result shows that signal from inputs possibly come faster to output neurons 1,2

than to neuron 3. Neurons past red line are possibly useless for left part of topology. On closer inspection I find that the red line progressing through binds only to the left direction. This means that the right part of this network graph is unnecessary. After deleting these neurons, the output of this network will remain the same. This algorithm enables us to simplify the unefficient network topologies without changing their behaviour.

5.8.2 Power Advanced Algorithm Hiearchical Example

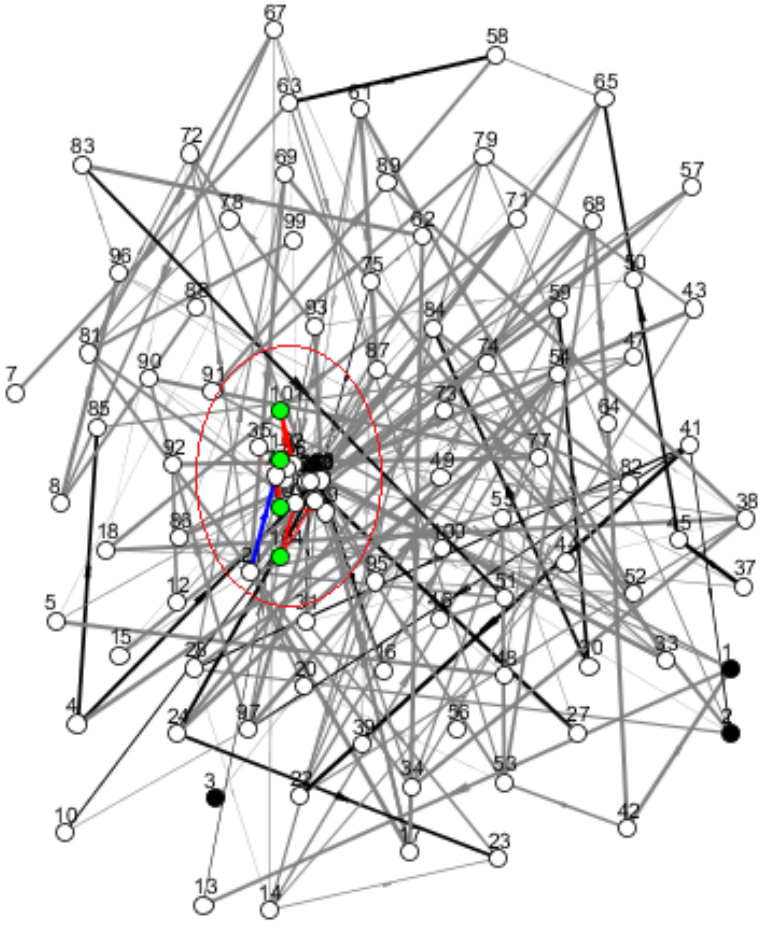

Figure 5.7: ***Advanced Power Algorithm*** *- algorithm works with imaginary power binds. Weight represents the probability of binding force of common shot both neurons in a defined window size shooting time. The resulting picture shows that the cluster in the center most likely fires together. The rest of neurons fires asynchronously or does not fire at all. We can say this because the rest of network has homogenous density of neurons in the space.*

Neurons in oval are the only active part of graph. Other neurons are placed equally into space. Algorithm tries to maximize distance to other neurons. Because this neurons are unbinded (they do not shoot) only criteria is the distance maximization. Distances are calculated only of the neurons closer than 100 pixels. In this case, the centre of the picture contains those neurons, which are critical for the network function. In the direction from the centre there are less and less useful neurons.

While the application supports exporting the list of selected neurons, it can be easily used for removing the unimportant neurons from the network, also called pruning of the ANN .

Prunning decreases the computation needs of the network and also increases its generalisation ability.

5.8.3 Comparison of Active algorithms Small Network

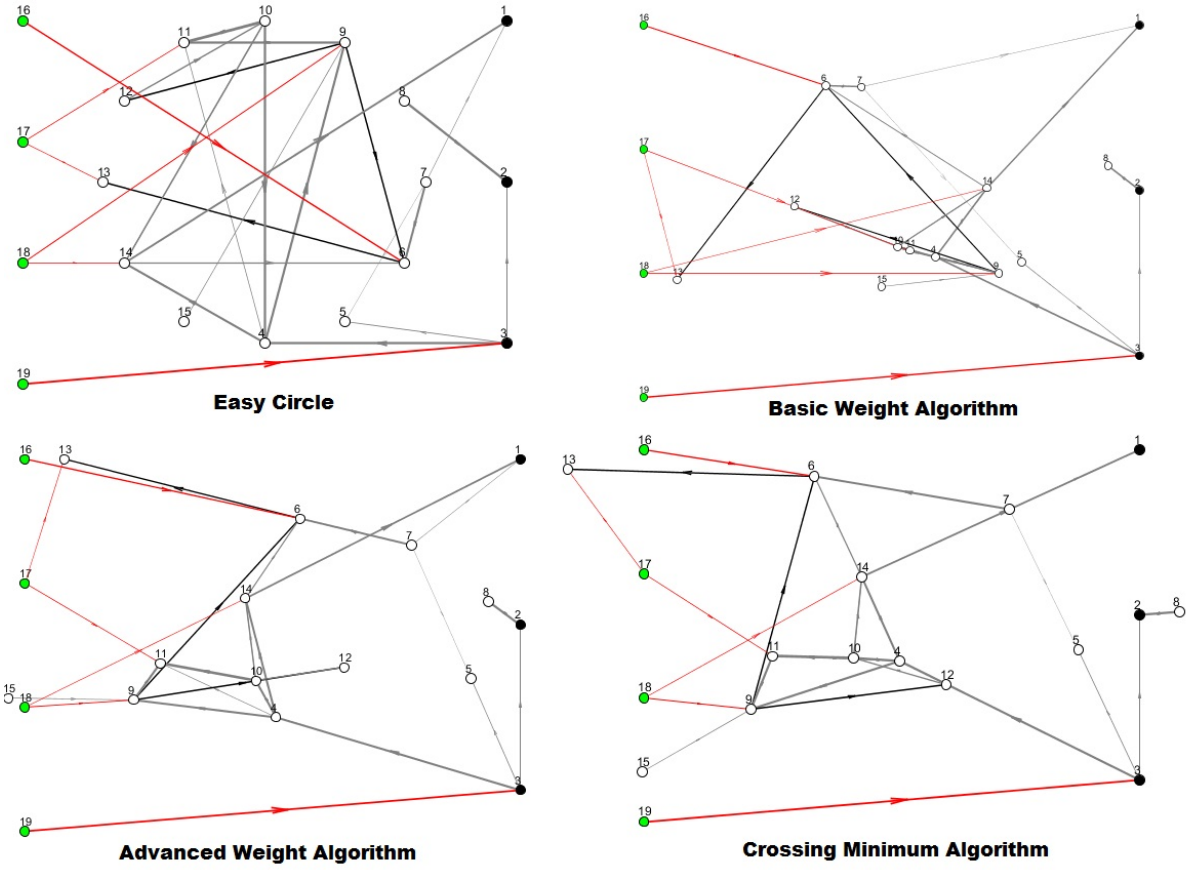

Figure 5.8: ***Comparison of active algorithms*** *- All algorithms were used on "Easy Circle" distribution of neurons and algorithms was studying real binds between neurons (not power). Tested topology has small number of neurons with normal number of binds. Original distribution on Fig. 4.3*

Here I try to comment results I got.

Basic weight algorithm (right top figure) (section 5.3) accepts only one criterion - bind weight. Algorithm tries to place binded neurons in distances based on weight. Neurons 6,7 are placed relatively well - aside from other neurons (they are weakly connected to rest of network). Neurons 4,9,10,11 are placed close together but result looks confused and unclear. We can see that this neurons are strongly connected together but binds between them are hardly resolvable.

Advanced weight algorithm (left bottom figure) (section 5.4) accepts mainly distances

related to weights and as second criterion tries to maximize distances between neurons. Compared to the basic algorithm, the situation around neurons 6 and 7, but a bit worse, however, the situation around the cluster of neurons 4,9,10,11 is greatly improved. Compared with low crossing is seen that the visibility decreases with increasing number of crossings. This algorithm have the best avoidance to placing binds almost parallel.

Crossing minimum (right bottom figure) (section 5.5) accepts mainly sum of crossing matrix and as second criterion distances related to weights. Distribution of neurons in space looks similar like advanced algorithm results. The results show that the algorithm placed neurons evenly spontaneously deployed to the area despite the fact that it does not force any criterion. Confusing situation happens about almost intersecting binds. As an example can be mentioned bind between neuronS 3 and 4 which is going through neuron 12.

5.8.4 Comparison of Active algorithms Fullbinded Network

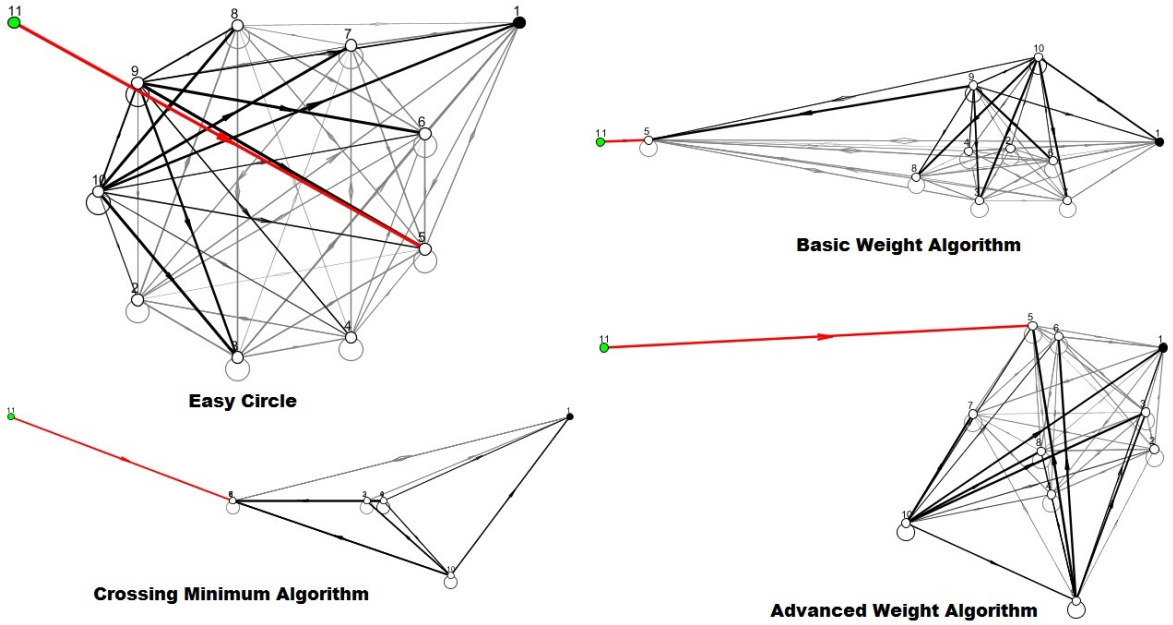

Figure 5.9: ***Comparison of active algorithms*** *- All algorithms were used on "Easy Circle" distribution of neurons and algorithms was studying real binds between neurons (not power). Tested topology has small number of neurons withalmost fully connected graph.*

Here I try to comment results I got.

Basic weight algorithm (right top figure) (section 5.3) - bind weight. The results show that neurons 9,10 are pushed from rest of neurons by inhibition binds. Because algorithm takes bind from the lightest to the strongest, it is neuron 5 is tightening closer to input neuron (repeater). Because of the high number of binds with similar weight distances between rest neurons change only slightly.

Advanced weight algorithm (right bottom figure) (section 5.4) accepts mainly distances related to weights and as second criterion tries to maximize distances between neurons. In comparison with basic algorithm, the algorithm respects better the sum of weights from binds connected on neuron 5 and this sum hold neuron with rest of internal neurons. Group of neurons 9,10 is separated from rest neurons too but in comparison with previous result two groups are not so clearly separated.

Crossing minimum (left bottom figure) (section 5.5) accepts mainly sum of crossing matrix and as second criterion distances related to weights. The result show that fully binded network reveals weakness of the algorithm. The algorithm tends to give bind parallel. The result has the minimum crossisng but the result is confusing.

5.8.5 XOR tested by algorithms

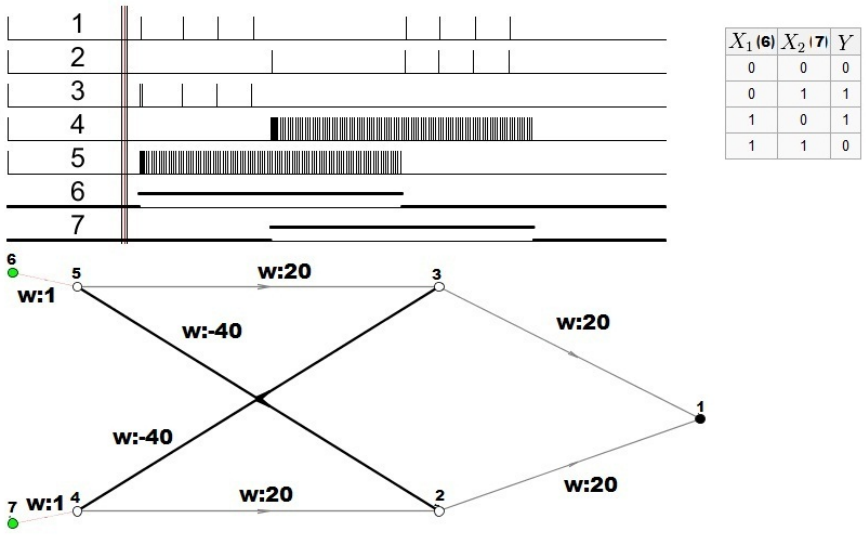

Figure 5.10: ***XOR dynamic context*** *- This network implements binary exclusive-or (XOR) function. The output of the function is true if and only if the values on inputs are different. From the fig. X we can see that binary values are coded using the value of analog current. Binary 0 is represented by 0mA and binary 1 is representd by 50mA. The output is coded by spikes. Logical zero corresponds to no spikes, and logical one corresponds to regular firing of output neuron.*

On this standard neural network representing XOR I try to explain the dynamic continuity in the formation of power binds.

On first part of spiketrain (where neuron 6 is at powered and neuron 7 has zero current) neuron 5 will be binded with neuron 1 and 3. These bind will not be much strong because neuron 5 got excited much more times than neurons 1 and 3. Neuron 3 and neuron 1 will be binded hard because excitation of neuron 3 cause excitation of neuron 1.

On second part of spiketrain (where both inputs are powered) will shoot only neuron 4 and 5 because they are powered from input. Between this neurons will be created moderate bind, because one shoot after another repeatly, even though no bind between them.

On third part is situation similar like in part one but excitated is bottom side of network. Binds will be created between neurons 4,2 and 1.

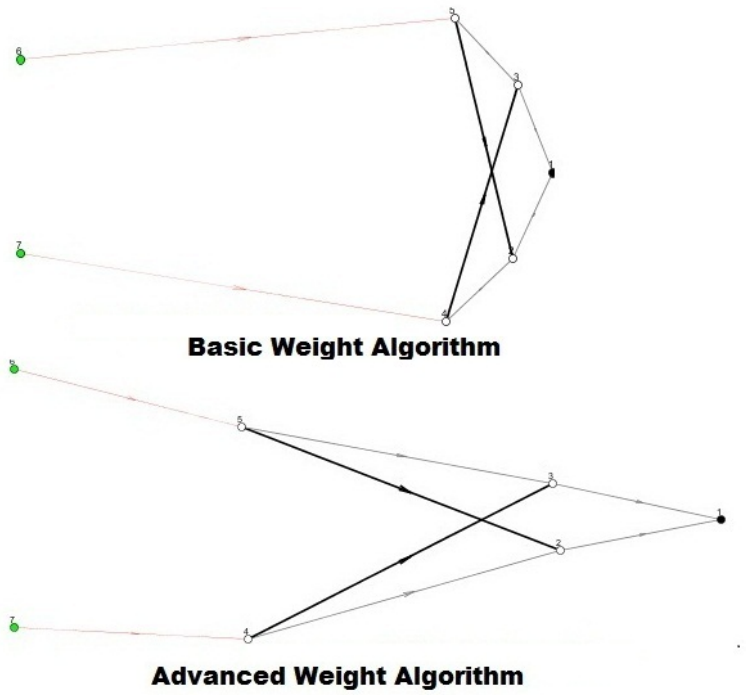

Figure 5.11: ***XOR results*** *- results of weight algorithms used on network Fig. 5.10.*

Here I try to comment results I got for weight algorithms.

Basic weight algorithm (section 5.3) shows that binds in network are few times stronger than binds which connect inputs with rest of network. The algorithm confirms that weights in network correspond with original shape.

Advanced weight algorithm (section 5.4) ignores ratio between weight of input binds and rest of the binds and tries to place rest of the binds to the best shape. The result also corresponds to the original shape of the distribution.

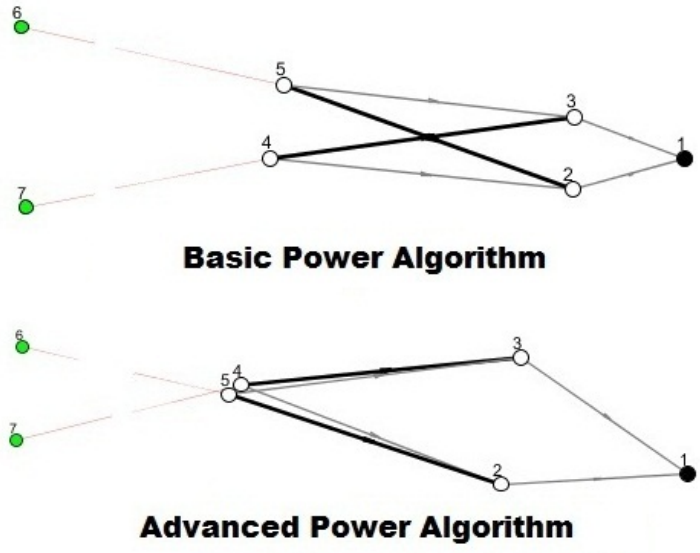

Figure 5.12: ***XOR results*** *- results of weight algorithms used on network Fig. 5.10.*

Here I try to comment results I got for power algorithms.

Basic power algorithm (section 5.3) results looks like original shape of network. The result suggests that the resulting bind strength ratios have similar size as the real links in the neural network, I mean that behaviour of this network is directly based on its binds.

Advanced power algorithm (section 5.4) can reveal less important regularities of spiketrain. Neurons 4,5 are close to each other. It is based on their power bind from middle part of spiketrain. Also this neurons switch places. This is due to the large difference in frequency of firing of these neurons and the rest of the network. These binds are weaker than other, less important power binds.

Chapter 6

Thesis Conclusion

Result of my work is the application able to draw and visualize neural network and its behavior. The application contains several tools and principles useful for normalisation of drawed form. In this way, we can study different kinds of ANN by single application and formulate results generally, for different types of networks.

The application tries to visualisate networks behaviour with the best quantum in every moment that the visualisation is displayed with the highest frame rate available for given simulation speed. I tried to access the animation and imaginatively to ensure minimal computational complexity of the best tools. In result, the application searches only in a narrow range of loaded data. Numerical complexity of visualisation grows lineary with increasing number of binds. For example, if all neurons fire at once, the number of pulses relative to the calculations is exactly equal to the number of binds despite the fact that all the pulses that travels in the animation are stored in single list.

In order to be able to study the patterns and characteristics of networks I implemented several methods. Each implemented algorithm respects different rules with different importance but all algorithms do it in order of clarity. The algorithms have their own numerical complexity. User can easily see this information on the displayed FPS . Passive algorithms calculate only single cycle of calculations. Their numerical complexity is for most networks minimal and is directly based on number of neurons. Active algorithms calculates up to 1000 cycles or can be stopped by user. Numerical complexity can play important role. It can be calculated as number of calculations.

Basic weight algorithm in each cycle calculates as many calculations as the number of binds (power or weight). Total number is equal to total number of binds b.

Advanced weight algorithm in each cycle also calculates all binds but also calculates distances between all neurons. Total number is equal to $(b + n^2) \cdot 9$. Where n is number

of neurons. Multiplied by 9 because calculation has to be done for each surrounding position.

Crossing minimum algorithm in each cycle calculates crossings with other binds for each bind twice (bind is connected to two neurons). Total number of calculation is $2 \cdot 9 \cdot b^2$.

Tested networks have number of binds is commonly 1-5 times higher than number of neurons for common networks. Maximal number of binds is restricted to 4000 (based on maximal size of memory). For network contained from approximately 100 binds and approximately 30 neurons, Basic Algorithm has complexity approximately 30 calculations per cycle, Advanced Algorithm has complexity approximately 9000 calculatiions per cycle and for Crossing Minimum is complexity per cycle approximately 180.000 times.

For maximal number of binds can grow complexity of Crossing Minimum to tens and houndreds milions of calculations.

The numbers are approximate (all types of calculations do not take same time) but show how complexity grows with the size of the network. For Basic Algorithm grows lineary, for advanced algorithm grows exponentially and multiplied by 9 and for Crossing Minimum grows faster exponentially (number of bind is commonly higer than number of neurons) and multiplied by 18.

This work aims to create a visualization tool that helps us somehow illuminate the internal structure and behavior of recurrent neural networks 3rd generation. This is to provide algorithms for 2D chart neuron in space where neurons are distributed based on the selected criterion.

Due to a lot of different properties of possible networks, I showed that different algorithms are suitable for different types of networks. Therefore, I design, implemented, tested and compared with each other several algorithms that optimize both static and dynamic properties of the site. I showed how algorithms can help us understand the internal structure of these networks and how this knowledge can be further exploited. Finally, I showed which algorithms are suitable for different networks.

Bibliography

Encog (2012), *Encog Artificial Intelligence Framework for Java* [online]. [cit. 2012-05-14], ⟨hhttp://code.google.com/p/encog-java/⟩.

Eugene M. Izhikevich (2003), *IEEE Transactions on Neural Networks* [online]. [cit. 2012-05-23], ⟨http://www.izhikevich.org/publications/spikes.html⟩.

Example Depot (2012), *Listing the Files or Subdirectories in a Directory* [online]. [cit. 2012-02-22], ⟨http://www.exampledepot.com/egs/java.io/GetFiles.html⟩.

FANN (2012), *Fast Artificial Neural Network Library* [online]. [cit. 2012-05-14], ⟨http://leenissen.dk/fann/wp/⟩.

Gephi (2012), *The Open Graph Viz Platform* [online]. [cit. 2012-05-14], ⟨http://gephi.org/⟩.

Gerstner and Kistler (2012), *4.1 Integrate-and-fire model* [online]. [cit. 2012-05-14], ⟨http://icwww.epfl.ch/ gerstner/SPNM/node26.html#Fig2-IF⟩.

Graphviz (2012), *Graph Visualization Software* [online]. [cit. 2012-05-14], ⟨http://www.graphviz.org/⟩.

Guess (2012), *The Graph Exploration System* [online]. [cit. 2012-05-13], ⟨http://graphexploration.cond.org/⟩.

L.R. Varshney, B.L. Chen, E. Paniagua, D.H. Hall and D.B. Chklovskii (2012), *Neuronal Connectivity II* [online]. [cit. 2012-04-15], ⟨http://www.wormatlas.org/neuronalwiring.html⟩.

Maass, W. (1996), *Networks of Spiking Neurons: The Third Generation of Neural Network Models (1996)* , In Journal Neural Networks vol. 10, p. 1659–1671.

NeuroML (2012), *Current applications* [online]. [cit. 2012-05-14], ⟨http://www.neuroml.org/index.php⟩.

Neuroph (2012), *Java Neural Network Framework* [online]. [cit. 2012-05-12], ⟨http://neuroph.sourceforge.net/⟩.

Piccolo2D (2008*a*), *A Structured 2D Graphics Framework* [online]. [cit. 2012-05-12], ⟨http://www.piccolo2d.org/⟩.

Piccolo2D (2008*b*), *Graph Editor* [online]. [cit. 2012-02-22], ⟨http://www.piccolo2d.org/learn/grapheditor.html⟩.

University of Maryland (2008), *Piccolo Toolkit* [online]. [cit. 2012-05-14], ⟨http://www.cs.umd.edu/hcil/piccolo/index.shtml⟩.

Vítků, J. (2011), *An Artificial Creature Capable of Learning from Experience in Order to Fulfill More Complex Tasks*, Diploma Thesis, Czech Technical University in Prague, Faculty of Electrical Engineering.

Vítků, J. (2012*a*), '*Parser Ver0.3 - readtopology.m* [dvd]'.

Vítků, J. (2012*b*), '*Parser Ver0.3 - writespiketrain.m* [dvd]'.

Wikipedia (2012), *Wikipedia - Graph theory* [online]. [cit. 2012-05-22], ⟨http://en.wikipedia.org/wiki/Graph theory⟩.

yWorks (2012), *yEd Graph Editor* [online]. [cit. 2012-05-14], ⟨http://www.yworks.com/en/index.html⟩.

Appendix A

Contents of CD

This Bachelor Thesis includes the compact disc with the following content:

- Electronic version of this document in PDF format.
- Project containing the source code in the Java language implementing visualisation tool.
- Project containing the source code in the Matlab language implementing parser for simulation neural networks and exports data in predefined format

Appendix B

User Manual and Implementation notes

In this chapter I would like to briefly describe how to launch the selected experiment. The following section will describe the particular window of aplication GUI and its basic purpose.

B.1 Launching the Experiments

All of parts necessary for launching the experiment are implemented in the Java programming language, because of this fact the entire simulation should be totally platform independent. Launching of the experiments has been tested on the platforms: Windows, partly on Linux and OS X.

B.2 Manual

1. Box with names of topology which will be loaded.
2. Left click opens file browser where can user choose file. Right click browses default directory.
3. Box with names of spiketrain which will be loaded
4. Left click opens file browser where can user choose file. Right click browses default directory.

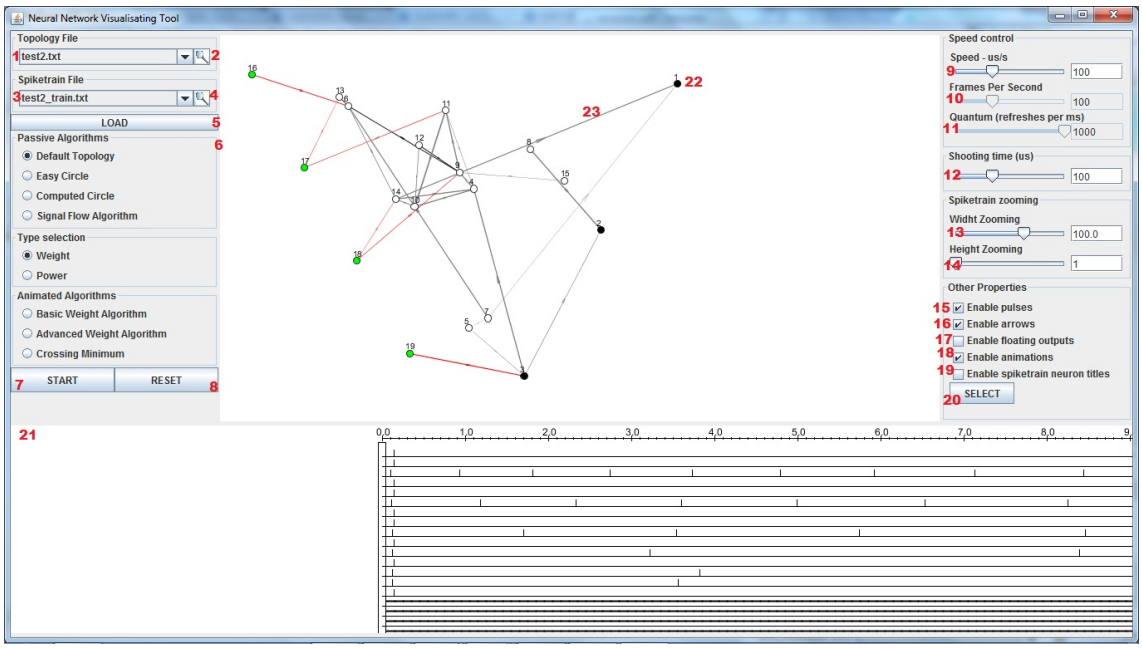

Figure B.1: ***Application Appearance*** *- appearance of the visualizationtool displaying simple testing network.*

5. Button which loads selected networks. When application loading all tools are disabled.
6. Chosen algorithm will be independently applicated on the network in current state.
7. Button starts the animation. If animation or algorithm is running, can pause both.
8. Reset network to default distribution and reset the time to zero.
9. Sets speed of animation. Sized in microseconds.
10. Shows FPS of animation or animated algorithm.
11. Shows actually set value of quantum.
12. Sets wide of time window and affects parameter of power algorithms.
13. Sets horizontal zoom of spiketrain. Sized to pixels per milisecond.
14. Sets vertical zoom of spiketrain. Constrain are resized if neuron titles are enabled.
15. Enables animation of pulses.
16. Enables drawing of arrows.
17. Enables floating of outputs in algorithms.
18. Adds additional delay for each cycle of animated algorithms.
19. Enables showing neuron titles in spiketrain and their refreshing.

20. After click network is frozen and user can identify any part of the graph. After confirm click selected neuron are displayed and saved in text file select.txt in root of the project.
21. If animation is paused user can change time by dragging spiketrain.
22. If user clicks neuron, it shows its information. If user exits the neuron or information field, information disappears.
23. If user enters bind, it shows its information. If user exits the neuron or information field, information disappears.

B.3 Source Data Examples

```
1  0.02 0.2 -65 8 30
2  2
3  3
4  5
5  5 5 0.02 0.2 -65 8 30 2 -0.1 1 3 0.2 2
6  4.5 6 0.02 0.2 -65 8 30 3 0.3 3 5 0.13 1 4 0.4 4
7  5 7 0.02 0.2 -65 8 30
8  2 6 4 0.44 0.55 4 -0.44 0.86
9  4 2
10 1.5 1 1 -0.9 0.1 5 -0.1 0.1 4 1 1
11 1 2 2 0.99 0.01 4 0.6 1 3 0.97 1
```

Figure B.2: ***Topology File*** *- grey rectangle contains numbers neurons, red one contains abcd treshold parameters, in green one are some x,y coordinates and blue one contains information about bind - to(4), weight(0.44) and delay (0.55).*

```
1  2
2  3
3  5
4  0.105377 1.095559 2.217459 3.516828 5.071157 7.025070 9.693733
5  0.102855 1.003932 2.008403 3.146192 4.463129 6.035446 8.001305
6  0.103404 1.022637 2.050495 3.219028 4.577905 6.211789 8.278000
7  0.088561 0.668261 1.282982 1.938163 2.639192 3.392690 4.207871
8  0.145543
9
10 0.1
11 0 75.000 75.000 75.000 75.000 75.000 75.000 75.000 75.000 75.000
12 0 75.000 75.000 75.000 75.000 75.000 75.000 75.000 75.000 75.000
```

Figure B.3: ***Spiketrain File*** *- grey rectangle contains numbers neurons, red one contains one time of excitation neuron ID 1, in blue one is analog parameter, black one indicates that neuron 6 is analog and yellow one contains value of one current step.*