Resource Discovery in Peer-to-Peer Networks Using Evolutionary Neural Networks

Annemari Auvinen, Hermanni Hyytiälä, Heikki Kainulainen,
Niko Kotilainen, Anu Niemi, Joni Töyrylä and Mikko Vapa

Version 1.0, May 30, 2003

University of Jyväskylä
Department of Mathematical Information Technology

Abstract

Resource discovery is an essential problem in peer-to-peer networks as there is no centralized index where to look for information about resources. One solution for the problem is to use a search algorithm that locates the resources based on the local information about the network. Traditionally, these search algorithms have been designed to be based on few rules and designed by humans. The problem with these algorithms is that if the conditions in the network change the algorithm becomes less efficient and won't have the flexibility to adapt to the new environment.

In this document we describe the results of a process where evolutionary neural networks were used for finding an efficient search algorithm. By using this process we can define the network conditions and the quality of the algorithm and let the computer find the solution for us. The initial test results indicate that without a prior knowledge about good search algorithm an evolutionary optimization process can produce candidates that are better compared to traditional search algorithms used in loosely-structured peer-to-peer networks. This work was made as an exercise assignment for TIE340 Artificial Intelligence -course and serves as a basis for further study about the subject.

Contents

1 Introduction **1**
- 1.1 Research Process 2
- 1.2 Division of Tasks 3

2 Peer-to-Peer Networks **4**
- 2.1 Brief History 4
- 2.2 Peer-to-Peer Network Models 4
  - 2.2.1 Loosely Structured 4
  - 2.2.2 Tightly Structured 5

3 Research Problem **6**
- 3.1 Breadth-First Search Algorithm 6
- 3.2 Neural Network Algorithm - NeuroSearch 6

4 Neural Network Optimization Process using Gaussian Random Variation **10**

5 Test Case **12**

6 Results and Analysis **14**

7 Conclusion **20**

Bibliography **22**

Appendix 1. NeuroSearch Code **23**

Appendix 2. NeuroGenerator Code **27**

Appendix 3. Test Case Code **33**

Appendix 4. The Best Evolved Neural Network **34**

1 Introduction

Evolutionary processes have become common in information technology research because they provide simple means for global optimization which is often needed when solving complex problems [17]. One such problem is resource discovery in peer-to-peer networks in which the total effect of local decisions in the nodes produces global outcome that might not be easily figured out just by thinking. Therefore, we sought out to find a solution for a complex problem using a flexible algorithm structure that could adapt to different conditions.

Feedforward neural network [13] (also called as Multi-Layer Perceptron, MLP) is known as one example of flexible algorithm structure. This is because if there are enough neurons in the neural net it can work as an universal approximator. Universal approximator feature means that if the structure is properly taught neural network can learn to map any function that is given as a training data. In our case we didn't knew beforehand the right answers, so we couldn't use traditional teaching methods where all the input/output pairs are known beforehand. Thus we had to rely on evolution and competition.

Evolutionary algorithms (such as genetic algorithm) use a process where initial population is instantiated as candidate solutions and then refined in forthcoming generations using mutation and crossover. Traditionally crossover has been the fundamental part of the variation algorithm, but in recent years its benefits have been questioned [12]. Especially in cases where there is no clear building blocks that form the final solution crossover might not be useful. This is also the case in neural networks where crossing two good networks over won't usually produce a good offspring. Because of this we decided to include only the mutation operation and checked how far it might bring us.

Teaching problems in presence of no training data for neural networks can be avoided if there is only a sufficient information to rank the candidates. Generally this means that we don't need exact output values of the neural network but only some measurement that tells whether one neural net is better than the other. This could be easily determined by measuring the collective effect of neural network decisions. In peer-to-peer networks this collective effect might for example be the number of packets the algorithm used and the number of results the algorithm was able to retrieve. In fact this was all that we needed for the system to be working.

1.1 Research Process

Our research process consisted of five distinct phases. First the whole concept had to be thought out which included the reformulation of resource discovery problem for neural networks and finding out a proper optimization mechanism for the problem. Then we had to gather a group that would be large enough to have the whole system running in short time. This was quite easy because there were people wanting to take part of Artificial Intelligence course and also the course's exercise work. After forming the group tasks were divided based on everyone's background and skills so that the process would go as smoothly as possible. These tasks included:

- definition of neural network and its inputs,
- definition of fitness function for ranking the neural networks,
- definition of variation and selection process to be used,
- implementation of NeuroGenerator class for managing the neural networks,
- implementation of NeuroSearch class for testing the algorithm in peer-to-peer node,
- definition and implementation of test case,
- conducting and analysing the test case and
- writing the report.

As a test platform we had an implementation of Chedar peer-to-peer node which has been designed by Cheese Factory -project [5] in Agora Center. For illustrating the behavior of peer-to-peer network also an user interface Peer-to-Peer Studio had been designed. Chedar uses TCP-sockets for communication and performs all the message handling operations in XML. This posed a problem for our work because executing one search takes a lot more time than would be needed so at the middle of the experiment we came into conclusion that a simulator might be needed. So we implemented a new simulator application using parts of Chedar's code and the user interface from Peer-to-Peer Studio and made the whole optimization process approximately 1000 times faster than with the TCP implementation.

Another problem that we faced was the stupidity of neural network. When randomly instantiated the neural network search algorithm produces candidates that might not forward messages at all and those that can't stop flooding the network. The first case is easy because it makes no harm to the system and gets only a small fitness. But the latter case is more problematic because we cannot move to evaluate the next neural network before the earlier one has stopped. We solved this problem by forcing the algorithm to stop flooding after 300 packets and rewarded him with a below one fitness. This made sure that the optimization process could continue and would more easily find out candidates that are more sparing.

Now when the system was running we got the first results how neural network is performing and found out that breadth-first search (BFS) is going to lose the competition. Initial results suggested that neural network is saving at least 20 % of packets compared to BFS. To improve the results we introduced three new input neurons to tell the neural network what is the neighbor degree, how many percentage of queries the neighbor has replied to and whether the node has already received the message from the neighbor.

Then we defined the final test case, implemented the scenario using the simulator and gathered the results from both neural network and breadth-first search algorithm. After the analysis this report was written and the results documented.

1.2 Division of Tasks

Tasks were equally divided among the group. Annemari made the implementation of NeuroSearch algorithm and defined the neural network structure and documented them with Anu. Hermanni wrote the introductionary peer-to-peer networks part and defined the resource discovery problem as graph search problem. Heikki implemented the NeuroGenerator and analysed the test case results. Niko implemented the simulator, defined the fitness function for neural networks and implemented the test case. Joni studied the backgrounds of the optimization problem, defined the variation and selection process and implemented the first version of variation function. Mikko conceptualized the problem and pushed the process on the finishing line. All the members were involved in writing the final report.

2 Peer-to-Peer Networks

2.1 Brief History

The Internet was originally established in the late 1960s [15]. The objective of the ARPANET-project was to share computers' resources among military computers around the United States. The most challenging purpose of ARPANET was to integrate different kinds of existing network technologies with one common network architecture. The ARPANET connected the first few hosts together not in client/server relationship, but rather as equal networking *peers*. This could be seen as the starting point of both the Peer-to-Peer concept and the Internet [18].

The most popular form of modern Peer-to-Peer computing is file-sharing. In this scenario, participants of Peer-to-Peer network share their file resources with other participants. This is a form of distributed file system (e.g., [16]). A modern Peer-to-Peer system is composed of an *application* level overlay network, i.e., network operates at the application level and forms a logical network overlay on top of physical network with regard to the ISO-OSI reference model (e.g., [19]). Compared to ARPANET's Peer-to-Peer functionality, modern Peer-to-Peer systems are *ad hoc*, i.e., peers join and leave the system constantly. Thus, this property poses challenges for efficient construction and maintenance of the overlay network and performing efficient data lookups in a distributed environment.

2.2 Peer-to-Peer Network Models

There are two approaches in which all modern Peer-to-Peer systems fall: the loosely structured approach and the tightly structured approach [11]. By structure, we refer to the topology of the overlay network, i.e., how the connections between participating peers are created and maintained. Next, we will briefly cover the key properties of these approaches.

2.2.1 Loosely Structured

In the loosely structured approach the construction and the maintenance of the overlay is controlled loosely. The placement of services and the topology of overlay is random. The data lookup model in loosely structured systems is not very efficient, because of unstructured properties of the overlay. Data lookup model is a combination of methods which are used for locating data in the overlay. Gnutella [10] is a well-known example of loosely structured overlay system. Gnutella is a pure Peer-to-Peer network as no peer is more important than any other peer in the network. The construction and maintenance of Gnutella network is extremely ad hoc, since participating peers can form the overlay network based on *local* knowledge (i.e., a peer has no knowledge of global state of the system). The Gnutella network can be considered as a variation of power-law graph [2]. In power-law graphs only few peers have high number of neighbor connections and the majority of peers have low number of neighbor connections.

2.2.2 Tightly Structured

Partly due to scalability problems of loosely structured systems, several tightly structured overlays have been proposed. In the tightly structured approach, the overlay is constructed deterministically, which all participating peers have to follow; the topology of the overlay and the placement of services is controlled tightly. With tightly structured systems, it is feasible to perform *global* data lookups in the overlay efficiently. By global lookup, we mean that the system is able to find a service from the overlay, if it exists. While there are significant differences among proposed tightly structured systems, they all have in common that *peer identifiers* are assigned to participating peers from a large *identifier space* by the overlay. Globally unique identifiers are also assigned to application-specific data items, *keys*, which are selected from the same identifier space.

3 Research Problem

In the *resource discovery problem* any node in the network can possess resources and also query these resources from other nodes. The problem consists of graph with nodes, links and resources. Resources are identified by IDs and nodes can contain any number of resources. There can also be duplicates of the same resource in different nodes. One node knows only the resources it is hosting therefore all the other nodes must find out where queried resources are located. Any node in the graph can start a query which means that some of the links in the graph are traversed based on a local decision and whenever the query reaches the node with the queried ID, the node replies. Figure 3.1 illustrates the resource discovery problem.

3.1 Breadth-First Search Algorithm

One possible solution for the resource discovery problem is the breadth-first search algorithm (BFS). In BFS the node that starts a query passes the query to all its neighbors. When the neighbors receive the query, they pass it further to all their neighbors except the one from which the query was received. Also if the query has already arrived to the node earlier it is not flooded further anymore. The query ends when there is no link that the query has not passed. This solution is illustrated in Figure 3.2.

3.2 Neural Network Algorithm - NeuroSearch

Breadth-first search algorithm ensures that if resource is located in the network it can also be found from the network. The drawback of the algorithm, however, is that it uses lots of query packets to find the needed resources. Therefore we designed an alternative algorithm that would overcome this problem.

NeuroSearch makes decision to whom of the node's neighbors the resource request message is forwarded based on the output of the neural network described in Figure 3.3. The resource reply message is forwarded back to the neighbor which forwarded the request to the node. Forwarding the query can be stopped by giving the ID of the message to the algorithm.

When resource request arrives to the algorithm it goes through all the node's neighbor connections one by one with the neural network. The query

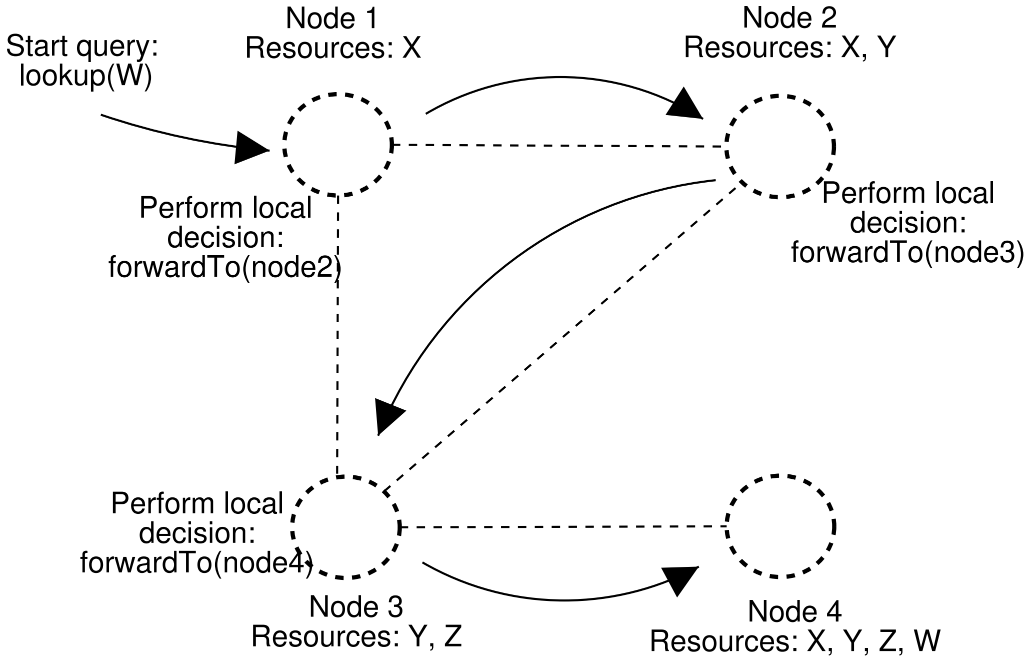

Figure 3.1: Resource discovery problem

includes network's weights.

The input parameters for the neural network are:

- *Bias = 1* is the bias term.
- *Hops* is the number of the hops in the message.
- *NeighborsOrder* tells in which neighbor rank this connection is compared to others. The connection with best rank has the value of 0.
- *Neighbors* is the number of the connection's neighbors.
- *Sent* has value 1 if the message has already been forwarded to the connection. Otherwise it has value of 0.
- *Received* has value 1 if the message came to the node from the connection, else it has value of 0.

Hops and NeighborsOrder are scaled with the function

$$f(x) = \frac{1}{x+1}, \tag{3.1}$$

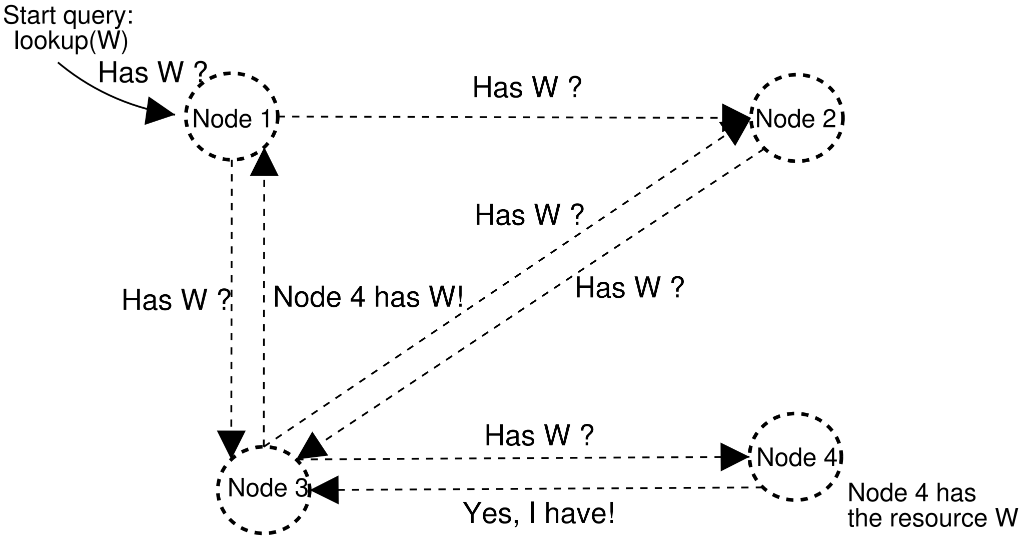

Figure 3.2: The breadth-first search represented as a graph model. In this figure, there are four steps which are involved in the process of finding a specific resource item in the graph. Step 1: Node 1 starts a lookup process for resource W and sends the query to all its neighbor nodes. Step 2: The neighbors of node 1 forwards the query to their neighbors respectively. Step 3: Node 4 replies to node 3 that it has the resource W. Step 4: Node 3 tells to node 1 that node 4 has the resource W.

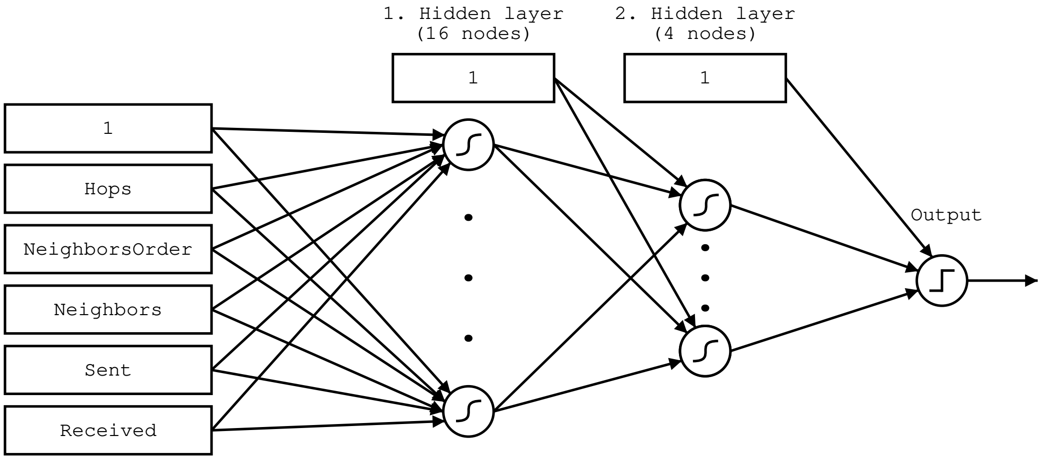

Figure 3.3: Neural network for NeuroSearch

and Neighbors with

$$f(x) = 1 - \frac{1}{x+1} \tag{3.2}$$

before giving them to the network. Scaling is made to ensure that all the inputs are between 0 and 1.

There are two hidden layers in the network. In the first hidden layer there are 15 nodes and in the second 3. Tanh is used as activation function in the hidden layers:

$$t(a) = \frac{2}{1 + \exp -2 * a} - 1. \tag{3.3}$$

Activation function in the output node is the threshold function:

$$s(a) = \left\{ \begin{array}{l} 0, x < 0 \\ 1, x \geq 0 \end{array} \right. . \tag{3.4}$$

If the output is 1, the message is forwarded to the connection.

4 Neural Network Optimization Process using Gaussian Random Variation

Our assumption is that we do not know which weights will make good neural net. To find the appropriate weights we imitate the evolution where stronger ones will survive. The decision, which neural nets are better than the others, can be done for example by counting the packets sent to network and replies received from it using following formula:

$$fitness = \frac{replies}{packets} \tag{4.1}$$

Higher fitness will prove better chance of survival to pass to the next round and lower fitness value indicates that search is not working good enough. Poorest neural nets will not send any packets further or they send every packet flooding the whole network thus resulting near zero fitnesses.

Optimization process must have an initial population e.g., 100 neural nets whose weights are randomly defined. Next every neural net will be tested at the peer-to-peer network and fitness value is calculated. When all neural nets have been tested we can choose for example 50 best as survivals of this generation of evolution. The surviving 50 neural nets are then used to breed new generation of neural nets. Everyone of those is copied and mutated and after this process we will again have 100 neural nets ready for testing the new generation.

Mutating neural net means that the values of the weights are slightly changed from the original. This is done by random variation using normal distribution (also called as Gaussian distribution).

At first we made Gaussian random variation function using free "OR-objects" probability package from DRA Systems [8]. There was normal distribution class which was easy to use. We just needed to give new mean and standard deviation to object and then receive wanted random number from normal distribution.

However, David Fogel and Kumar Chellapilla did variation differently in their research [6, 7, 9] by using scaling factor for each weight. Scaling factor improves the adaptability of the evolutionary search so we wanted our neural network to also have this feature. By following their example in our work we changed the variation function. The newer version of random variation function looked like this:

$$\sigma_i'(j) = \sigma_i(j) exp(\tau N_j(0,1)), j = 1, ..., N_w \tag{4.2}$$

$$w_i(j)' = w_i(j) + \sigma_i'(j) N_j(0,1), j = 1, ..., N_w \tag{4.3}$$

where N_w is the total number of weights and bias terms in the neural network, $\tau = 1/sqrt(2sqrt(N_w))$, $N_j(0,1)$ 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.

5 Test Case

In the test case we used power-law graph generated by Barabasi-Albert model [4, 3]. Power-law networks have a property that neighbour distribution follows power-curve ($P(k) = \frac{1}{k^\gamma}$, where γ for Barabasi-Albert graph is 3). Because of neighbor distribution there exists few hubs in the network that have many neighbors and lots of nodes that have only few links. The graph in the test case was generated to contain 100 nodes with the highest degree node having 25 neighbors. This graph topology is illustrated in Figure

Resources are distributed such a way that there are 25 different resources in the network and the number of different resources in a node is the same as the number of neighbors the node has. This means that the largest hub has one copy of all resources and the lower degree nodes different resources randomly chosen from uniform distribution.

All the nodes have equal probability to be query originators (randomly sampled from uniform distribution) and 50 queries are executed for each neural net in a generation. The sequence of 50 queried resources is prefixed and always the same. The probability for a resource i to be selected from all resources as next queried resource is $P(i) = \frac{\Sigma i}{\Sigma n}$ where n is the total number of resources. This means that more common resources will be queried more often and the querying order is same, but the query originators change in each generation. This ensures that we are not overteaching neural networks to adapt to only one environment, but rather to changing circumstances.

The fitness for a neural network is defined in two parts. Each query j is *scored* for the neural network and the $fitness$ is calculated by summing up all the scores after 50 queries. Mathematically speaking the fitness for neural network h is $fitness_h = \Sigma_{j=1}^{50} score_j$.

The score is defined with four rules:

1. If $replies \geq \frac{\Sigma i}{2}$ then $score = 50 \times \frac{\Sigma i}{2} - packets$
2. If $replies < \frac{\Sigma i}{2}$ then $score = 50 \times replies - packets$
3. If $packets > 300$ then $score = 1 - \frac{1}{packets+1}$
4. If $replies = 0$ then $score = 1 - \frac{1}{packets+1}$

The first rule ensures that if enough resources are found from the network the fewer packets the neural network uses the better. The second rule ensures that if number of found resources is not enough then the neural network gets better if it gains more resources than others. The third rule was made as an addition to avoid flooding the network infinitely. In that case an algorithm that stops is always better than algorithm that does not. The fourth rule makes sure that if none of the resources are found then neural network should increase the number of packets sent to the network.

30 neural networks were used in the test case and 15 neural networks selected after each generation for next round. 15 new offsprings were created by using variation equations 4.2 and 4.3 for computing new weights. As a stopping criteria for the optimization process we set 5.000 generations which seemed to be taking approximately 20 hours (less than a day).

After the optimization process had finished we selected as a best algorithm the one that is the most aged among the best 15 neural networks that succeeded to the last round. By selecting the oldest neural network we ensured that the network has survived many generations and proved that it is worth to be used for testing against Breadth-First Search algorithm.

6 Results and Analysis

To evaluate the difference between BFS and NeuroSearch we calculated the number of packets used and received replies for 50 different queries used in the test case. The results are illustrated in Figures 6.1, 6.2 and 6.3.

As can be seen from Figure 6.1 BFS floods the whole network of 100 nodes always producing 297 query packets. NeuroSearch is much more efficient resulting in average of 60 packets per query. The reason why there is some variation in the number of packets for successive NeuroSearch queries is that the distance where packets will be delivered depends on which node is querying. If the query starts from a central node it will produce more packets than the same query started from an edge node.

Figure 6.2 shows how many replies both of the algorithms produce. BFS locates all instances of a queried resource so these replies also represents the maximum number that might have been found in each query from the network. NeuroSearch in the other hand aims to locate only half of the resources in the network because of its fitness function and therefore does not find the resources that would have required extra query packets.

For each query NeuroSearch locates approximately half of the resources which can be seen from Figure 6.3. There are 6 queries in which NeuroSearch misses the target to locate half of the resources. In 2 queries two resources are missed and in 4 queries one resource is missed. This indicates that optimization process has found an algorithm that with a small error can locate half of the resources from the network saving a significant number of packets and thus reducing the generated network load.

With evolutionary algorithms it may be hard to determine whether the optimization process has reached the near optimal solution or would a longer evolution process produce better candidates. By viewing the evolution curve shown in Figure 6.4 we assume that because after 1000 generations the fitness has neither increased nor decreased the optimization process had reached a stable state.

The early phases of evolution (50 generations) are illustrated in Figure 6.5. Graph shows that the evolution process locates good candidates early in the process (approximately in 15th generation). Because we used 30 neural networks and 15 of them were varied in each generation the total variations needed for 15th generation was 225. The behavior that a good candidate is found early in the process is common in evolutionary algorithms so the results are as expected. Also because the optimization process does not seem

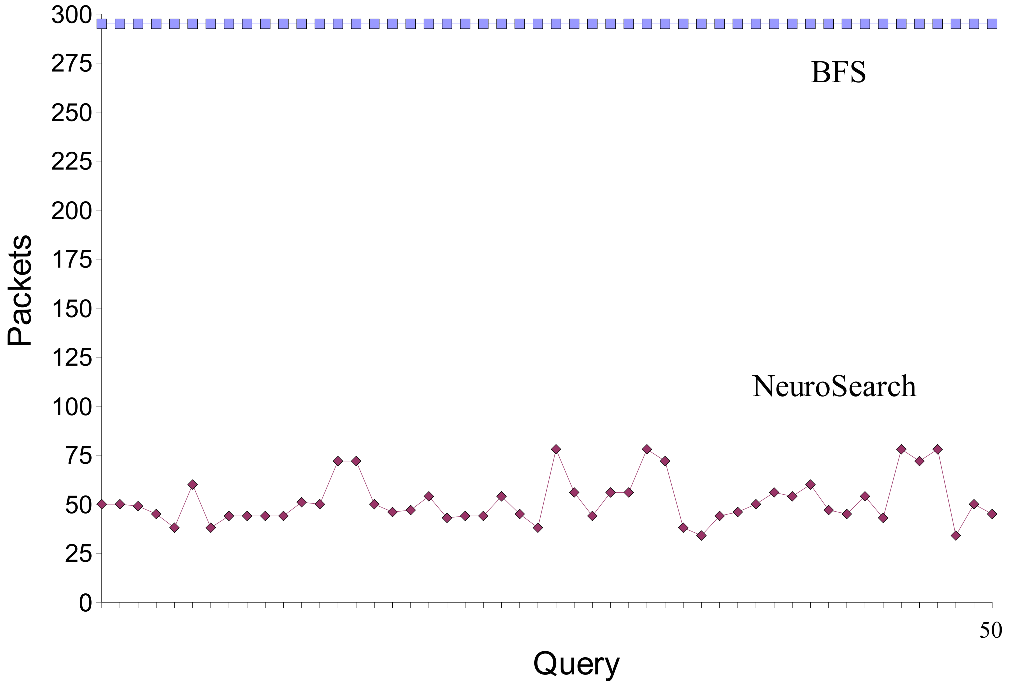

Figure 6.1: Packets

to take so much time to converge adding more input neurons might be a good idea for improving NeuroSearch's performance.

After drawing the graphs we briefly analyzed the behavior of the best evolved neural network shown in Appendix 4 by varying the input values of each input neuron and checking how it would affect the output. Increasing *Hops* resulted that the output would decrease. This is quite obvious by intuition because the longer the query travels the higher should be the dropping probability for the packet. *Sent* and *Received* had also the effect that when they were set to 1 the query was less likely to be forwarded. This is because usually there isn't a reason to forward the query twice the same link.

The interesting point came out when we analyzed the behavior of *NeighborsOrder* and *Neighbors*. For searching in power-law graphs it has been suggested that a good strategy to locate resources is to first send the query to highest degree nodes and then gradually lower the degree when searching proceeds [1, 14]. This would make sure that if resources are likely to be found from the network core the query finds them early in the process. For designing such a network the higher degree nodes should be preferred to lower degree ones suggesting that *NeighborOrder* in our case should be also high (the highest degree node gets value 1). But this was not true. The like-

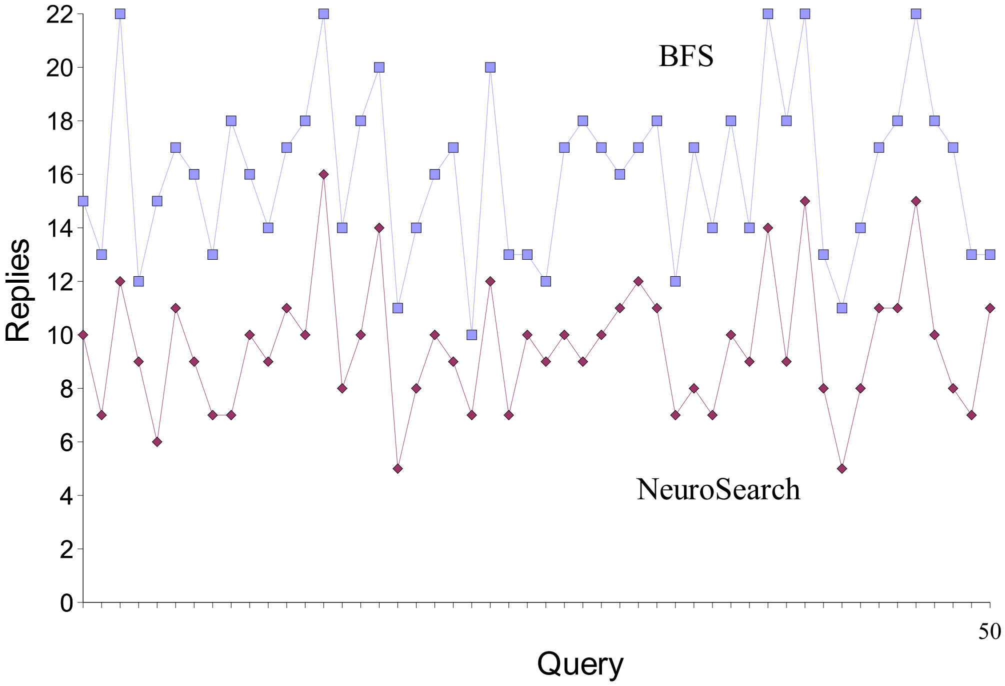

Figure 6.2: Real replies

lihood to forward the query increased when *NeighborsOrder* decreased. The same phenomena happened with *Neighbors* where fewer number of neighbors resulted in bigger output value of the neural net.

The explanation that we came up with is quite simple. Because central nodes in the network are highly connected they have a bigger probability to get the query from many directions. Therefore it is sufficient for only one link to get through to the central node and the other incoming queries may be dropped. This means that there is no reason to forward the query to the central nodes because they may already have got it from somewhere else. Therefore a search strategy which avoids central points might be better than the ones preferring them.

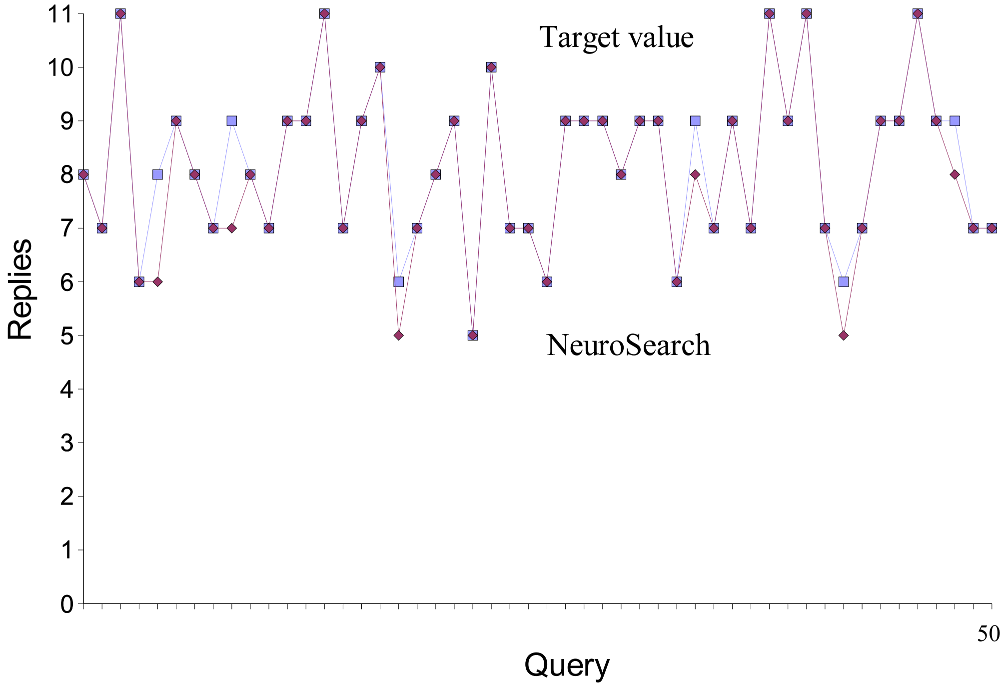

Figure 6.3: Replies

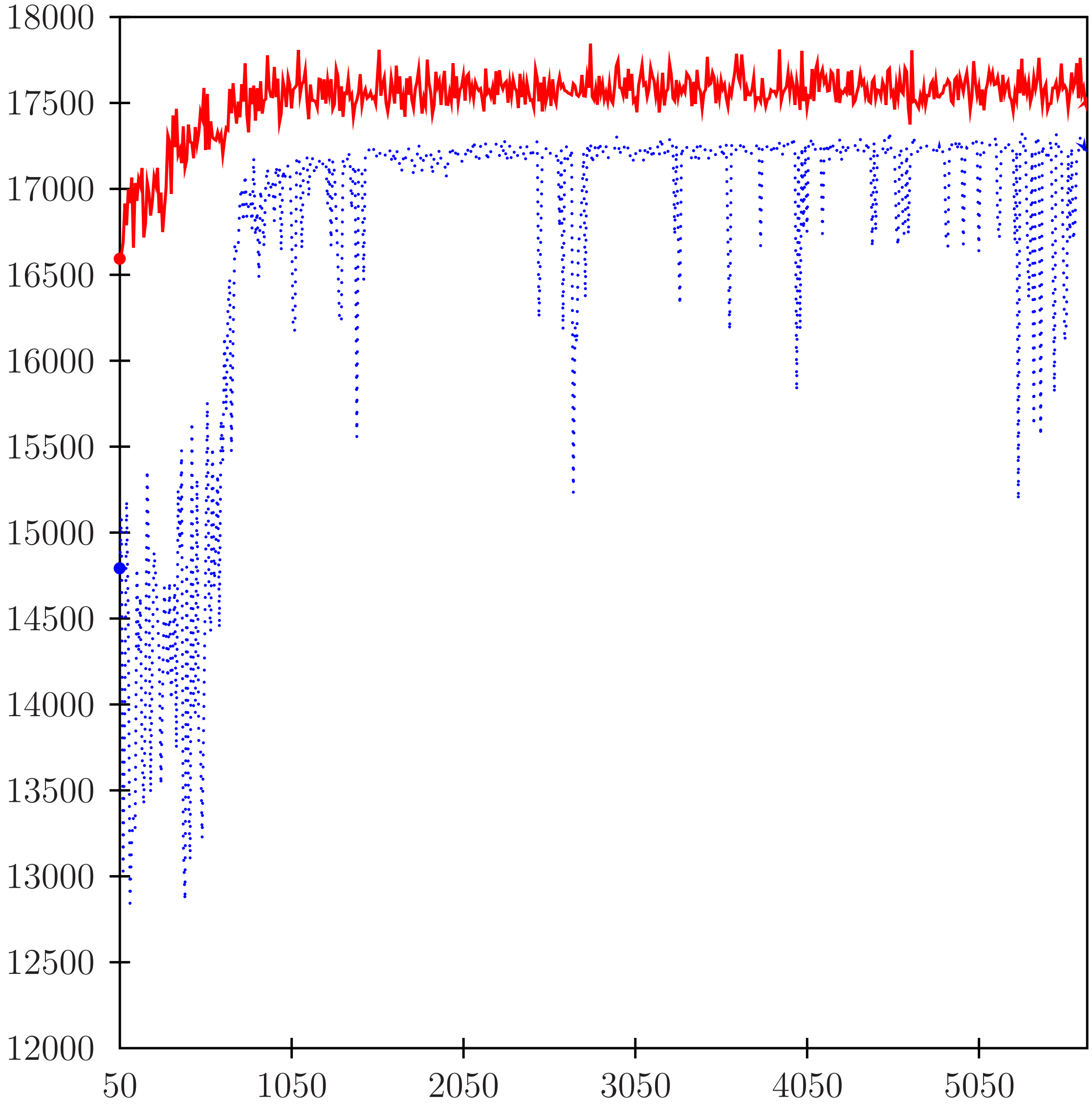

Figure 6.4: Evolution from 50th generation

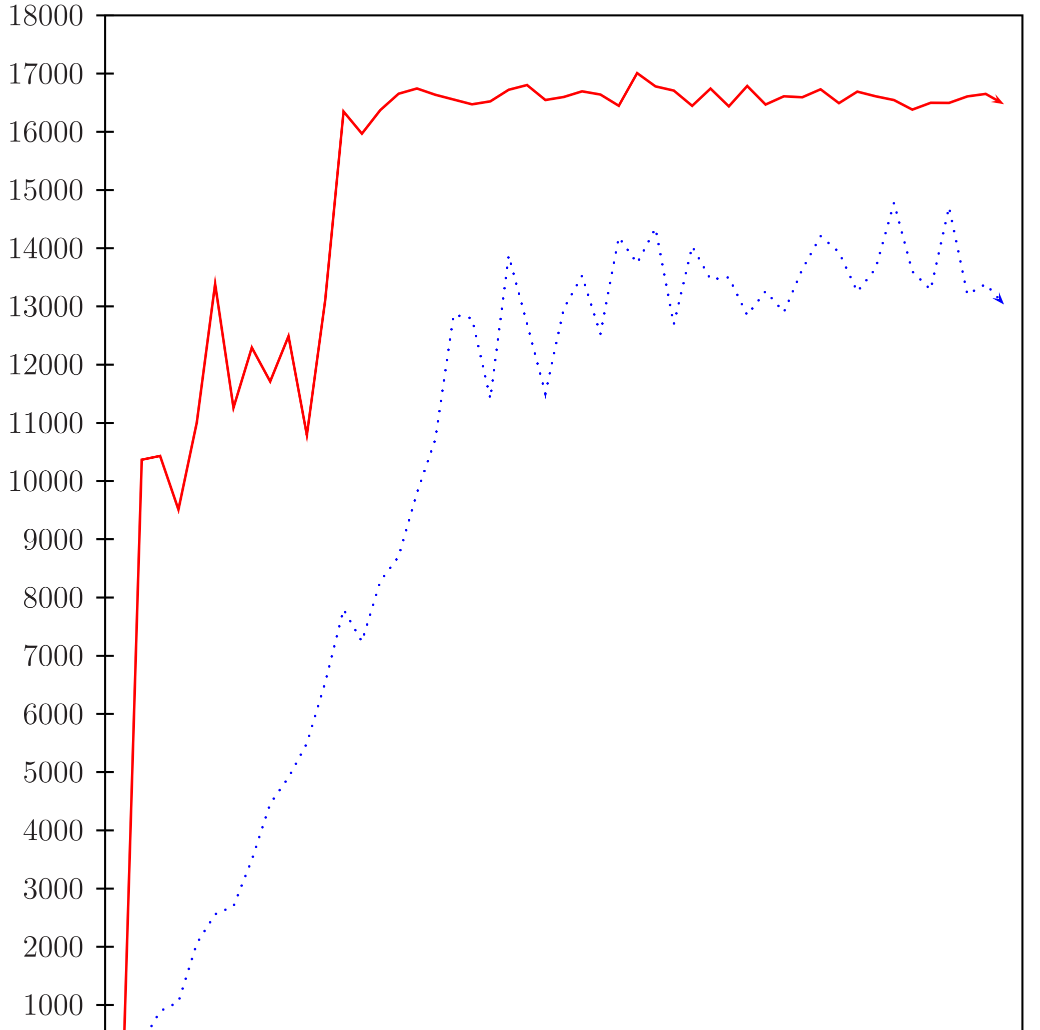

Figure 6.5: Early phases of evolution

7 Conclusion

We have showed that a properly designed neural network can find a good search algorithm using very simple optimization method. The neural network takes into account the special characteristics of its environment and therefore can be used in many different kind of peer-to-peer networks. Also because this process does not need human intervention the resulting algorithm is not limited to the designer's own ideas but only to the limitations of neural network's structure and its input neurons.

The behavior of the best evolved neural network was analyzed briefly and already showed some contradiction between earlier research results. An interesting thing might be to research this topic further to gain understanding how power-law structure and resource distribution is linked together.

Also there are many directions that were left out of this research. For example is there an algorithm that would perform well in dynamic conditions where topology is changing, what are the scalability factors of NeuroSearch, what improvements would be gained by adding more input neurons e.g., the number of replied results and what effects of changing the goal (fitness function) would have. These are interesting questions to answer and surely require more extensive analysis on NeuroSearch.

Bibliography

[1] Lada A. Adamic, Rajan M. Lukose, Amit R. Puniyani and Bernardo A. Huberman. “Search in power-law networks”. *Physical Review E*, 64, 2001.

[2] R. Albert and A. Barabasi. “Statistical mechanics of complex networks”. *Reviews of Modern Physics*, 74 (47), 2002.

[3] Albert-László Barabási. *Linked: The New Science of Networks.* Perseus Publishing, 2002. ISBN 0738206679, 256 pp.

[4] Albert-László Barabási and Reka Albert. “Emergence of scaling in random networks”. *Science 286*, (pp. 509–512), 1999.

[5] “Cheese Factory”. http://tisu.it.jyu.fi/cheesefactory.

[6] Kumar Chellapilla and David Fogel. “Evolving neural networks to play checkers without relying on expert knowledge”. *IEEE Transactions on Neural Networks*, 10 (6), pp. 1382–1391, 1999.

[7] —. “Evolving an expert checkers playing program without using human expertise”. *IEEE Transactions on Evolutionary Computation*, 5 (4), pp. 422–428, 2001.

[8] “DRA-systems”. http://opsresearch.com/OR-Objects/.

[9] David B. Fogel. *Blondie24: Playing at the edge of AI.* Morgan Kaufmann, 2001. ISBN 1558607838, 424 pp.

[10] “Gnutella”. http://www.gnutella.com.

[11] Hermanni Hyytiälä. *Fenfire in Peer-to-Peer Environment.* Master’s thesis, University of Jyväskylä, March 2003.

[12] Ankit Jain and David B. Fogel. “Case studies in applying fitness distributions in evolutionary algorithms. ii. comparing the improvements from crossover and gaussian mutation on simple neural networks”. In “Proceedings of the 2000 IEEE Symposium on Combinations of Evolutionary Computation and Neural Networks”, (pp. 91–97). IEEE Press, 2000.

[13] Tommi Kärkkäinen. “Mlp-network in a layer-wise form: Derivations, consequences and applications to weight decay”. *Reports of the Department of Mathematical Information Technology Series C. Software Engineering and Computational Intelligence*, 2000.

[14] Beom Jun Kim, Chang No Yoon, Seung Kee Han and Hawoong Jeong. “Path finding strategies in scale-free networks”. *Physical Review E*, 65, 2002.

[15] Barry M. Leiner, Vinton G. Cerf et al. “The past and future history of the internet”. *Communications of the ACM*, 40 (2), pp. 102–108, 1997. ISSN 0001-0782.

[16] Eliezer Levy and Abraham Silberschatz. “Distributed file systems: concepts and examples”. *ACM Computing Surveys (CSUR)*, 22 (4), pp. 321–374, 1990. ISSN 0360-0300.

[17] Kaisa Miettinen, Marko Mäkelä, Pekka Neittaanmäki and Jacques Périaux (eds.). *Evolutionary Algorithms in Engineering and Computer Science.* John Wiley & Sons, Ltd, 1999. ISBN 0471999024, 256 pp.

[18] Andy Oram (ed.). *Harnessing the Power of Disruptive Technologies.* Sebastopol, CA: O’Reilly, 2001. ISBN 059600110X, 432 pp.

[19] R. Popescu-Zeletin. “Implementing the iso-osi reference model”. In “Proceedings of the eigth Data Communications Symposium”, (pp. 56–66). 1983.

Appendix 1. NeuroSearch Code

```
package netsimulator;

import java.util.*;

/**
 * <p>Title: NeuroSearch</p>
 * @author Annemari Auvinen, Anu Niemi, Niko Kotilainen
 * @version 1.1
 * @param nodes1 Number of nodes in the first hidden layer.
 * @param nodes2 Number of nodes in the second hidden layer.
 * @param inputAmount Amount of inputs.
 */
public class NeuroSearch {
  static final int nodes1 = 16;
  static final int nodes2 = 4;
  static final int inputAmount = 6;
  static final int weightAmount = inputAmount*nodes1+nodes1*nodes2+nodes2;

  /**
   * Handles the resource message. The method checks if the node is
   * providing queried resources and adds a reply if it does. Then the method
   * makes the forwarding decision based on the output of the neural network.
   * If output is 1 the message is forwarded to that connection.
   * @param node Node which is sending the message.
   * @param msg Message to handle
   * @return  Collection of Node objects where the message should be forwarded
   * to.
   */
  public static Vector handleMessage(Node node, Message msg) {
    Vector receivers = new Vector();

    if(msg.from != -1 && !node.received)
      if (node.hasResource(msg.resource))
```

```
      msg.replies++;

    Node connections[] = node.neighbors;

    if (msg.weights.length != weightAmount) {
      Common.error("Wrong amount of weights:" + msg.weights.length
                   + " should be " + weightAmount);
      return receivers;
    }

    LinkedList neighbors = node.getConnectionsOrderByNeighbors();
    for(int i=0; i<connections.length; i++) {
      Node con = connections[i];
      if (con.number == msg.from)
        continue;
      Vector data = inputData(node, con.number, msg, neighbors.indexOf(con),
                    con.neighbors.length);
      int output = calculateOutput(data, msg.weights);
      if(output == 1) {
        receivers.add(con);
      }
    }
    return receivers;
  }

  /**
   * Calculates the output of neural network for given data and weights. There
   * are 2 hidden layers and one output layer in the network. The activation
   * function in the hidden layers is the tanh function and in output threshold
   * function.
   * @param data  Input data
   * @param weights Weight values
   * @return  The output of neural network which is 1 or 0.
   */
  public static int calculateOutput(Vector data, double weights[]) {
    double layer1[] = new double[nodes1];
    double layer2[] = new double[nodes2];
    double sum;
    int j, i;

    //layer 1
    layer1[0] = 1.0;
    for(j=0; j<nodes1-1; j++) {
```

```
  sum = 0.0;
  for(i=0; i<data.size(); i++) {
   Double value = (Double)data.elementAt(i);
   double d = value.doubleValue();
   sum += d*weights[j*data.size()+i];
  }
  layer1[j+1] = tanh(sum);
 }

 //layer2
 int start = layer1.length*data.size();
 layer2[0] = 1.0;
 for(j=0; j<nodes2-1; j++) {
  sum = 0.0;
  for(i=0; i<layer1.length; i++) {
   double d = layer1[i];
   sum += d*weights[start +j*layer1.length+i];
  }
  layer2[j+1] = tanh(sum);
 }

 //output
 sum = 0.0;
 start = layer1.length*layer2.length;
 for(i=0; i<layer2.length; i++) {
  double d = layer1[i];
  sum += d*weights[start + i];
 }

 return thresholdFunction(sum);
}

/**
 * Threshold activation function
 * @param a Parameter of the function
 * @return  The value of the function
 */
public static int thresholdFunction(double a) {
 return a>=0 ? 1 : 0;
}

/**
 * Gets the input data for the connection.
```

```
   * @param node Node, which is sending the message.
   * @param neighOrder tells in which neighbor rank this connection is compared
   * to others. The connection with best rank has value 0.
   * @param connection Connection, where node is sending the message.
   * @param msg Message which is being queried.
   * @param neighRank Neighbor rank.
   * @param neighAmount Amount of neighbors.
   * @return Vector of input data
   */
  public static Vector inputData(Node node, int connection, Message msg,
                                 int neighRank, int neighAmount) {
    Vector data = new Vector();
    int hops = msg.hops;
    boolean sent = node.isForwardedTo(connection);
    data.add(new Double(1.0));
    data.add(new Double(scaleFunction(hops)));
    data.add(new Double(scaleFunction(neighRank)));
    data.add(new Double(1-scaleFunction(neighAmount)));
    data.add(sent ? new Double(1.0) : new Double(0.0));
    data.add(node.received ? new Double(1.0) : new Double(0.0));
    return data;
  }

  /**
   * The tanh function
   * @param a Parameter of the function
   * @return Value of the function
   */
  public static double tanh(double a) {
    return 2.0/(1+Math.exp(-2*a))-1;
  }

  /**
   * Scale function for input data
   * @param value Parameter of the function
   * @return Value of the function
   */
  public static double scaleFunction(int value) {
      return 1.0/(value+1);
  }

}
```

Appendix 2. NeuroGenerator Code

```
package netsimulator;

import java.io.*;
import java.util.*;

public class NeuroSearchGenerator{

    private NeuralNet[] net;

    private int numWeights;

    public NeuroSearchGenerator(int num, int numWeights){
        net = new NeuralNet[num];
        for(int i = 0; i < net.length; i++){
            net[i] = new NeuralNet(numWeights);
        }
        this.numWeights = numWeights;
    }

    public int size(){
        return net.length;
    }

    public NeuralNet get(int index) {
        return net[index];
    }

    public void variate(int generation) {
        Arrays.sort(net);
        int k = 0;
        for(int i = (net.length - 1); i >= (net.length / 2); i--){
```

```
            net[k] = new NeuralNet(net[i], generation);
            k++;
        }
    }

    public void write(String fileName, boolean append, boolean writeShort,
                      String divider) throws IOException{
        PrintWriter out = new PrintWriter(new BufferedWriter
                          (new FileWriter(fileName, append)));
        out.println(divider);
        for(int i = 0; i < net.length; i++){
          if(writeShort)
            out.println(net[i].toShortString());
          else
            out.println(net[i]);
        }
        out.close();
    }

}


package netsimulator;

import java.util.*;

public class NeuralNet implements Comparable{

    private static Random rand = new Random();

    private double[] weight;

    private double[] scale;

    private int packets;

    private int replies;

    private int realReplies;

    private int generation = 0;

    public int variationRound = 0;
```

```
private boolean evaluated = false;

private boolean stopped = false;

public NeuralNet(int num){
    weight = new double[num];
    scale = new double[weight.length];
    for(int i = 0; i < weight.length; i++){
        weight[i] = randomWeight(-0.2, 0.2);
        scale[i] = 0.05;
    }
}

public NeuralNet(NeuralNet father, int variationRound){
    weight = new double[father.weight.length];
    scale = new double[father.scale.length];
    generation = father.getGeneration() + 1;
    this.variationRound = variationRound;

    double avg = 0, big = 0, small=999;

    for(int i = 0; i < weight.length; i++){
      scale[i] = father.scale[i]
                 * Math.exp((1 / Math.sqrt(2 * Math.sqrt(weight.length)))
                 * gaussian(0,1));
        weight[i] = father.weight[i] + (scale[i] * gaussian(0,1));
    }
}

public void initState() {
  evaluated = false;
  stopped = false;
  packets = 0;
  replies = 0;
  realReplies = 0;
}

private static double gaussian(double mean, double deviation) {
    double gauss = rand.nextGaussian();
    gauss *= deviation;
    gauss += mean;
    return gauss;
}
```

```
private static double randomWeight(double min, double max){
    return ((max - min) * rand.nextDouble() + min);
}

public void increase(int pack, int repl, int realRepl){
  packets += pack;
  replies += repl;
  realReplies += realRepl;
  evaluated = true;
}

public double fitness(){
  if (!evaluated)
    return -1;

  if (replies == 0) {
    return 1-(double)1/(packets+1);
  }
  else
    return Math.max(replies*50-packets, 1);
}

public int compareTo(Object o){
  NeuralNet net = (NeuralNet)o;
  double d = net.fitness();
  if (fitness() < d)
    return -1;
  if (fitness() > d)
    return 1;
  if (net.generation > generation)
    return -1;
  if (net.generation < generation)
    return 1;
  return 0;
}

public String toQueryString() {
    StringBuffer sb = new StringBuffer();
    for(int i = 0; i < (weight.length - 1); i++) {
        sb.append(weight[i]);
        sb.append(';');
    }
```

```
    sb.append(weight[weight.length - 1]);
    return sb.toString();
}

public String toScaleString() {
    StringBuffer sb = new StringBuffer();
    for(int i = 0; i < (scale.length - 1); i++) {
        sb.append(scale[i]);
        sb.append(';');
    }
    sb.append(weight[weight.length - 1]);
    return sb.toString();
}

public String toShortString() {
    StringBuffer sb = new StringBuffer();
    sb.append("fitness:");
    sb.append(fitness());
    sb.append(" packets:");
    sb.append(packets);
    sb.append(" replies:");
    sb.append(replies);
    sb.append(" real replies:");
    sb.append(realReplies);
    sb.append(" gen:");
    sb.append(generation);
    sb.append("/");
    sb.append(variationRound);
    sb.append(" hash:");
    sb.append((toQueryString().hashCode()+toScaleString().hashCode()));
    return sb.toString();
}

public String toString() {
    StringBuffer sb = new StringBuffer();
    sb.append(fitness());
    sb.append('|');
    sb.append(generation);
    sb.append('|');
    sb.append(packets);
    sb.append('|');
    sb.append(replies);
    sb.append('|');
```

```
        sb.append(toQueryString());
        sb.append('|');
        sb.append(toScaleString());
        return sb.toString();
    }

    public double getWeight(int index){ return weight[index]; }

    public double[] getWeights(){ return weight; }

    public double getScale(int index){ return scale[index]; }

    public int getPackets(){ return packets; }

    public int getReplies(){ return replies; }
    public int getRealReplies(){ return realReplies; }

    public int getGeneration(){ return generation; }

    public void setWeight(int index, double d){ weight[index] = d; }

    public void setScale(int index, double d){ scale[index] = d; }

    public boolean getStopped() { return stopped; }

    public void setStopped(boolean stopped) {this.stopped = stopped; }

    public void setPackets(int i){
      packets = i;
      evaluated = true;
    }

    public void setReplies(int i){
      replies = i;
      evaluated = true;
    }

}
```

Appendix 3. Test Case Code

Appendix 4. The Best Evolved Neural Network

$w_{1,i\to j}$	$w_{2,i\to j}$	$w_{3,i\to j}$
$w_{1,1\to 1}$ 1.3247661425878028	$w_{2,1\to 1}$ -0.5387932824827377	$w_{3,1\to 1}$ -0.06371764660899448
$w_{1,2\to 1}$ -0.013466907192934821	$w_{2,2\to 1}$ -0.5355148540537485	$w_{3,2\to 1}$ 0.021425895678959382
$w_{1,3\to 1}$ -0.4770553213893772	$w_{2,3\to 1}$ 1.881183554640507	$w_{3,3\to 1}$ 3.40023955867343
$w_{1,4\to 1}$ -0.05942394679405162	$w_{2,4\to 1}$ 0.9921055719565186	$w_{3,4\to 1}$ -0.2181887102541971
$w_{1,5\to 1}$ -14.045504403892785	$w_{2,5\to 1}$ -0.15659692286680538	
$w_{1,6\to 1}$ -2.248171487089831	$w_{2,6\to 1}$ -0.3527033963108481	
$w_{1,1\to 2}$ -0.34528639463034166	$w_{2,7\to 1}$ 1.6247125365418191	
$w_{1,2\to 2}$ 0.40802052804753586	$w_{2,8\to 1}$ -0.689976075445654	
$w_{1,3\to 2}$ -0.06919500243719534	$w_{2,9\to 1}$ 0.22347261693586734	
$w_{1,4\to 2}$ 0.12400150070275985	$w_{2,10\to 1}$ -0.746667057890345	
$w_{1,5\to 2}$ 0.659573538880692	$w_{2,11\to 1}$ -0.6372266755512547	
$w_{1,6\to 2}$ -1.5884389048525815	$w_{2,12\to 1}$ -0.8938397224593093	
$w_{1,1\to 3}$ -6.852848348566475	$w_{2,13\to 1}$ -1.463134655416962	
$w_{1,2\to 3}$ 0.012572115300206486	$w_{2,14\to 1}$ -0.650425769067907	
$w_{1,3\to 3}$ 0.10194334242231132	$w_{2,15\to 1}$ 0.2596588519458894	
$w_{1,4\to 3}$ -0.6003001025565958	$w_{2,16\to 1}$ 8.794371250616374	
$w_{1,5\to 3}$ -0.38390645171303306	$w_{2,1\to 2}$ 0.853448285361938	
$w_{1,6\to 3}$ 0.1348173088416575	$w_{2,2\to 2}$ -0.21006494141858795	
$w_{1,1\to 4}$ 5.529157845244522	$w_{2,3\to 2}$ 0.22007361277057044	
$w_{1,2\to 4}$ 0.9537227660768783	$w_{2,4\to 2}$ 0.727819951808203	
$w_{1,3\to 4}$ 0.34583637952724433	$w_{2,5\to 2}$ 0.18425538642306902	
$w_{1,4\to 4}$ 0.09257807547234628	$w_{2,6\to 2}$ 0.03076584716299502	
$w_{1,5\to 4}$ -1.0694189574663793	$w_{2,7\to 2}$ -0.1877876200967109	
$w_{1,6\to 4}$ 0.31418412302495013	$w_{2,8\to 2}$ 0.416954342958482	
$w_{1,1\to 5}$ 0.7571450572445404	$w_{2,9\to 2}$ -0.07175396959179092	
$w_{1,2\to 5}$ 2.664434128382305E-4	$w_{2,10\to 2}$ 0.0802843811686893	
$w_{1,3\to 5}$ -0.0032780111165833375	$w_{2,11\to 2}$ -0.31086438543281275	
$w_{1,4\to 5}$ 0.25839322722297137	$w_{2,12\to 2}$ -1.5174802870172657	
$w_{1,5\to 5}$ 0.10552948162300968	$w_{2,13\to 2}$ -0.4205706069670075	
$w_{1,6\to 5}$ -6.130901200272487E-4	$w_{2,14\to 2}$ -0.0919420644708367	
$w_{1,1\to 6}$ -0.23108849324852382	$w_{2,15\to 2}$ -0.39226837544972204	
$w_{1,2\to 6}$ -2.7719289649020946	$w_{2,16\to 2}$ 0.18761285098262218	
$w_{1,3\to 6}$ -0.9508050335564902	$w_{2,1\to 3}$ -0.11549372787075549	
$w_{1,4\to 6}$ -0.07070127606136295	$w_{2,2\to 3}$ 0.5832624091135137	
$w_{1,5\to 6}$ -0.10548575300152353	$w_{2,3\to 3}$ -0.27461173212361556	
$w_{1,6\to 6}$ 0.1650429209742457	$w_{2,4\to 3}$ 1.4341592213787293	
$w_{1,1\to 7}$ 1.8220602755968627	$w_{2,5\to 3}$ -0.11032983607050127	
$w_{1,2\to 7}$ 2.7219277526609265	$w_{2,6\to 3}$ -0.11545839973441131	
$w_{1,3\to 7}$ -0.177607544963059	$w_{2,7\to 3}$ -0.018313774028569277	
$w_{1,4\to 7}$ 0.20551020646430565	$w_{2,8\to 3}$ 0.8091758163767527	
$w_{1,5\to 7}$ -0.15407497361991868	$w_{2,9\to 3}$ -0.01857331621914468	
$w_{1,6\to 7}$ 0.13400366021551624	$w_{2,10\to 3}$ 0.22690747603516886	
$w_{1,1\to 8}$ -0.4012324364445238	$w_{2,11\to 3}$ 1.2776427350969763	
$w_{1,2\to 8}$ 0.557678403489737	$w_{2,12\to 3}$ -0.1439622344428606	
$w_{1,3\to 8}$ 0.8101635261344102	$w_{2,13\to 3}$ -0.4942326588279249	
$w_{1,4\to 8}$ 0.42285396751992615	$w_{2,14\to 3}$ 0.25966834347438633	
$w_{1,5\to 8}$ -0.8273554282821771	$w_{2,15\to 3}$ 0.28407621875247374	
$w_{1,6\to 8}$ 0.03489316993633529	$w_{2,16\to 3}$ 0.2733774387298385	

$w_{1,i\to j}$	$w_{2,i\to j}$	$w_{3,i\to j}$
$w_{1,1\to 9}$ 0.13206449275504997		
$w_{1,2\to 9}$ -0.28772949553333077		
$w_{1,3\to 9}$ 0.11970904553081177		
$w_{1,4\to 9}$ 0.07855376190984534		
$w_{1,5\to 9}$ -0.008372607884388431		
$w_{1,6\to 9}$ -0.24011095749168004		
$w_{1,1\to 10}$ 4.6038255891402775		
$w_{1,2\to 10}$ 0.026563398979181675		
$w_{1,3\to 10}$ -24.653779702096866		
$w_{1,4\to 10}$ -0.6282130920707882		
$w_{1,5\to 10}$ -0.9699632127654465		
$w_{1,6\to 10}$ -0.053062671845711756		
$w_{1,1\to 11}$ -5.850790896816266		
$w_{1,2\to 11}$ 0.5281763771388885		
$w_{1,3\to 11}$ 8.870973451569046		
$w_{1,4\to 11}$ -0.010073566867611426		
$w_{1,5\to 11}$ -1.2680286248573345		
$w_{1,6\to 11}$ 0.6410864879949991		
$w_{1,1\to 12}$ 0.8803771462268215		
$w_{1,2\to 12}$ 3.4347501115947865		
$w_{1,3\to 12}$ -0.686426308098676		
$w_{1,4\to 12}$ -1.705934522845834		
$w_{1,5\to 12}$ -0.43025752090626973		
$w_{1,6\to 12}$ -0.0542126879636221		
$w_{1,1\to 13}$ -0.1255944409153011		
$w_{1,2\to 13}$ -0.15805055603745755		
$w_{1,3\to 13}$ -0.26897092517071886		
$w_{1,4\to 13}$ 2.0022081776628804		
$w_{1,5\to 13}$ 0.5861126432922974		
$w_{1,6\to 13}$ -0.22807788406433008		
$w_{1,1\to 14}$ -0.18589714304635951		
$w_{1,2\to 14}$ -0.02037802335852728		
$w_{1,3\to 14}$ 0.09549623308282254		
$w_{1,4\to 14}$ -1.5249394599461075		
$w_{1,5\to 14}$ -0.04818401553269423		
$w_{1,6\to 14}$ -1.8898868527342743		
$w_{1,1\to 15}$ 5.248141788055083		
$w_{1,2\to 15}$ -0.053856616699524566		
$w_{1,3\to 15}$ 0.14215358847329249		
$w_{1,4\to 15}$ 3.0622452315638364		
$w_{1,5\to 15}$ -0.0026269162438096877		
$w_{1,6\to 15}$ -1.0901013062122078		