Intelligent Peer-to-Peer Networks

Hermanni Hyytiälä, Niko Kotilainen, Joni Töyrylä and Mikko Vapa

Version 1.0, May 22, 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 use of evolutionary neural networks for finding an efficient search algorithm. By using neural networks 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 peer-to-peer networks.

Contents

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 [3]. 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 [2] (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 [1]. 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.

2 Peer-to-Peer Resource Discovery Problem

In the peer-to-peer *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 resource ID, the node replies. Figure 2.1 illustrates the resource discovery problem.

2.1 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 that the query has not yet traversed. The query ends when there is no link that the query has not passed. This solution is illustrated in Figure 2.2.

2.2 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 2.3. The resource reply message is forwarded back to the neighbor which forwarded the request to the node.

When resource request arrives to the algorithm it goes through all the

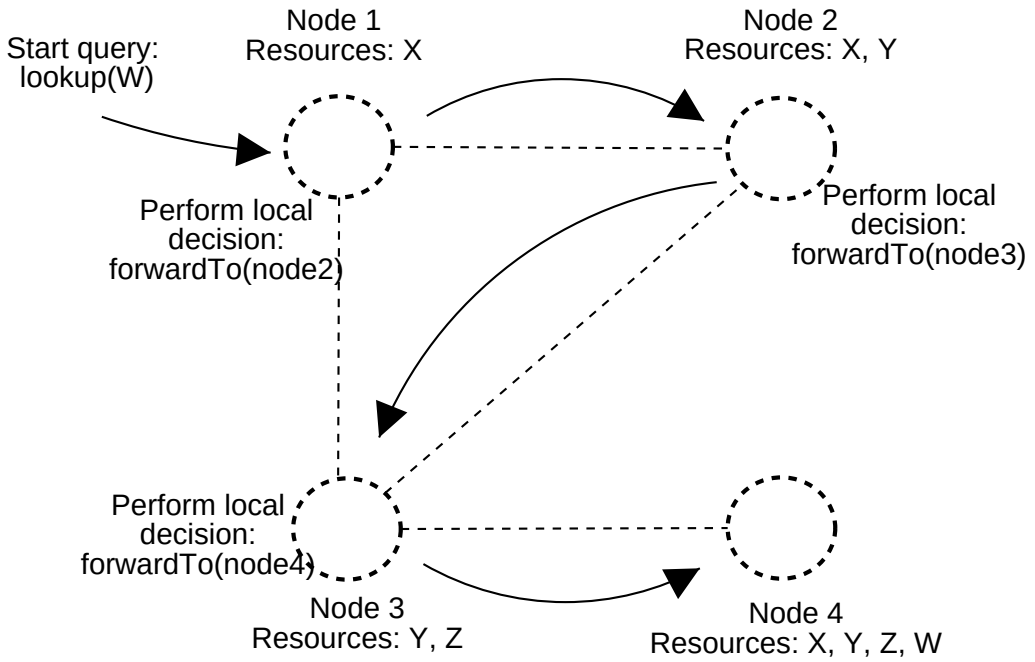

Figure 2.1: Resource discovery problem

node's neighbor connections one by one with the neural network. The query 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

$$f(x) = \frac{1}{x+1}, \tag{2.1}$$

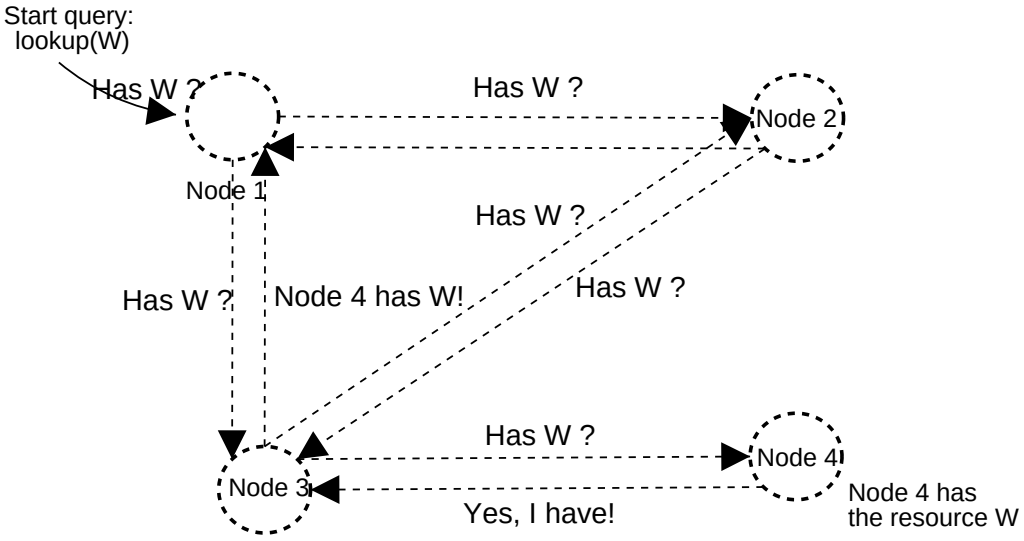

Figure 2.2: The breadth-first search represented as a graph model. In this figure, there are four steps which are involved in the process of finding a specific resource item in the graph. Step 1: Node 1 starts a lookup process for resource W and sends the query to all its neighbor nodes. Step 2: The 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.

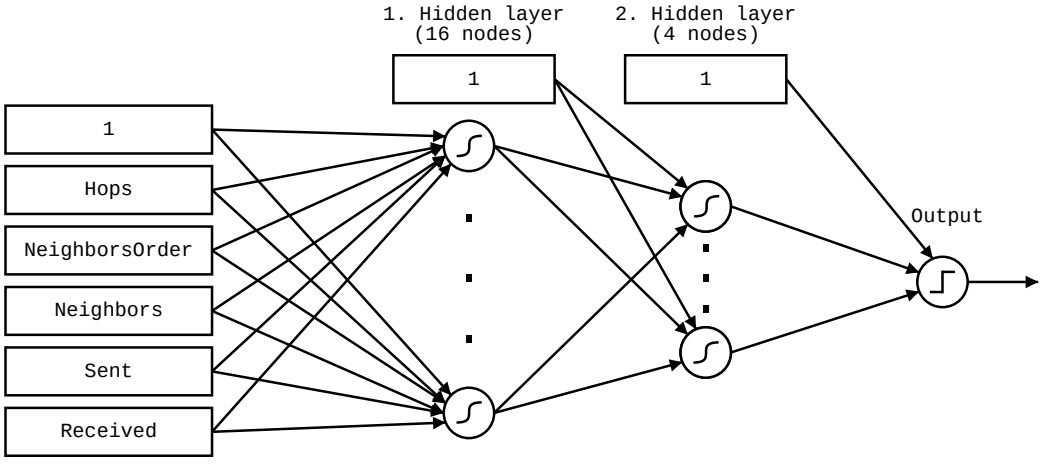

Figure 2.3: Neural network for NeuroSearch

and Neighbors with

$$f(x) = 1 - \frac{1}{x+1} \tag{2.2}$$

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:

$$t(a) = \frac{2}{1 + \exp -2 * a} - 1. \tag{2.3}$$

Activation function in the output node is the threshold function:

$$s(a) = \begin{cases} 0, x < 0 \\ 1, x \geq 0 \end{cases}. \tag{2.4}$$

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

3 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:

$$fitness = \frac{replies}{packets} \tag{3.1}$$

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). The variation function looks like this:

$$\sigma_i'(j) = \sigma_i(j) exp(\tau N_j(0,1)), j = 1, ..., N_w \tag{3.2}$$

$$w_i(j)' = w_i(j) + \sigma_i'(j) N_j(0,1), j = 1, ..., N_w \tag{3.3}$$

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.

Bibliography

[1] 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.

[2] 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.

[3] 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.