

Uppsala Master's Thesis in  
Computer Science  
Examensarbete DV3  
2002-08-30

# **Growing Neural Gas**

**Experiments with GNG,  
GNG with Utility and Supervised GNG**

Jim Holmström  
jimh@csd.uu.se

Uppsala University  
Department of Information Technology  
Computer Systems  
Box 337,  
SE-751 05 Uppsala,  
Sweden

Examiner: Olle Gällmo, crwth@docs.uu.se,  
Department of Information Technology, Computer Systems

## Abstract

This report aims to explain and experiment with the Growing Neural Gas algorithm, some problems are discussed and some modifications are suggested for future work. The GNG-U, which is a version of GNG that handles moving distributions is explained and experimented with as well. The SGNG or supervised-GNG algorithm for constructing Radial Basis Function Networks is explained and experimented with. A minor comparison between SGNG and GNG+RBF is performed and the result suggests that GNG+RBF performs equally well. A long-term problem with the SGNG algorithm is discussed and some improvements to SGNG are suggested.

<b>1 INTRODUCTION .....</b>	<b>1</b>
<b>2 GROWING NEURAL GAS.....</b>	<b>2</b>
2.1 INTRODUCTION TO GROWING NEURAL GAS .....	2
2.2 GROWING NEURAL GAS .....	3
2.2.1 GNG Pseudo-Code.....	4
2.3 GNG EXPLAINED.....	5
2.3.1 The Local Accumulated Error .....	5
2.3.2 Node Movements.....	6
2.3.3 Edges and the Induced Delaunay Triangulation .....	6
2.3.4 Node Insertion.....	6
2.4 EXPERIMENTS WITH GNG .....	8
2.5 SUMMARY AND DISCUSSION.....	12
<b>3 GNG WITH UTILITY FACTOR.....</b>	<b>13</b>
3.1 INTRODUCTION TO GNG WITH UTILITY FACTOR .....	13
3.2 GNG WITH UTILITY FACTOR.....	13
3.3 GNG-U EXPLAINED.....	14
3.3.1 The Utility Update Rule.....	14
3.3.2 Removal Criterion and Utility .....	14
3.3.3 Utility Initialisation for New Nodes .....	15
3.4 EXPERIMENTS ON GNG-U.....	15
3.5 SUMMARY AND DISCUSSION.....	18
<b>4 RADIAL BASIS FUNCTION NETWORKS.....</b>	<b>19</b>
4.1 INTRODUCTION TO RBF NETWORKS .....	19
4.2 TRAINING .....	20
<b>5 SUPERVISED GNG .....</b>	<b>21</b>
5.1 INTRODUCTION TO SUPERVISED GNG.....	21
5.2 SUPERVISED GNG .....	21
5.2.1 Brief Description.....	21
5.2.2 SGNG Pseudo-Code.....	22
5.3 SGNG EXPLAINED.....	24
5.3.1 The Local Accumulated Error .....	24
5.3.2 The Mean Distance and RBF Width .....	24
5.3.3 Insertion of New Nodes .....	24
5.3.4 Stopping Criteria .....	25
5.3.5 General Notes and Observations .....	25
5.4 EXPERIMENTS .....	25
5.5 SUMMARY .....	31
<b>6 IMPLEMENTATION .....</b>	<b>33</b>
6.1 INTRODUCTION .....	33
6.2 GENERAL DESIGN OVERVIEW.....	33

6.2.1 <i>The Components</i> .....	33
6.2.2 <i>Basic Elements</i> .....	34
6.2.3 <i>Input Generators</i> .....	34
6.2.4 <i>Algorithm Classes</i> .....	35
6.2.5 <i>Control Objects</i> .....	35
6.2.6 <i>Graphical Representation</i> .....	35
6.3 USER INSTRUCTIONS .....	35
<b>REFERENCES</b> .....	<b>38</b>

# 1 Introduction

Clustering can be described as the process of organizing a collection of  $k$ -dimensional vectors into groups whose members share similar features in some way. Each such group is represented by a  $k$ -dimensional vector called a code-vector (other names used are centre and node). The goal of clustering is to reduce large amounts of raw data by categorizing in smaller sets of similar items. The most common clustering algorithm is the K-means clustering algorithm by MacQueen [1], another algorithm widely used for vector quantization is Kohonen's Self Organizing Map (SOM) [2] and the Neural Gas algorithm described by Martinetz and Shulten [3].

In some cases little or no information is available about the input distribution or the size of the input data set, in these cases it is hard to determine a priori the number of nodes to use, such is the case in Kohonen's Self Organising Map and in the Neural Gas algorithm and also in classical K-means clustering.

This report describes in detail and explains an incremental clustering algorithm called Growing Neural Gas (GNG) by Bernd Fritzke. Numerous experiments are conducted on the GNG algorithm and later analysed. The GNG algorithm only has parameters that are constant in time and since it is incremental, there is no need to determine the number of nodes a priori. The issue of when GNG should stop is discussed shortly. Also described, discussed and experimented with are two derivatives of the GNG algorithm, the GNG-U and SGNG algorithms by the same author. GNG-U is a variation of GNG that is able to track non-stationary distributions by relocating less useful nodes, this is done by maintaining a local utility measure in each node, and performing relocations based on this measure. The final algorithm, SGNG, is presented in detail together with a short description of a Radial Basis Function network. SGNG can be described as a supervised variation of the GNG algorithm, a fusion of a slightly modified GNG with a Radial Basis Function (RBF) network. It is a method for building RBF networks in an on-line fashion, using the network squared-error to guide node insertions.

Finally, a short design overview of a C++ implementation of the three algorithms is presented together with a short user manual.

## 2 Growing Neural Gas

### 2.1 Introduction to Growing Neural Gas

In order to facilitate the understanding of the GNG algorithm, some basic concepts will first be introduced briefly. Firstly, clustering, the goal of which is to locate groups of similar data-items, and sometimes finding the number of groups or to cluster data-items into a predefined number of groups in the best possible way. Vector Quantization (VQ) is the process of quantizing  $n$ -dimensional input vectors to a limited set of  $n$ -dimensional output vectors referred to as code-vectors. The set of possible code-vectors is called the codebook. The codebook is usually generated by clustering a given set of training vectors (called training set), the codebook is then used to quantize input vectors.

The second and third concept is Voronoi diagrams and the Delaunay triangulation illustrated in figure 2.1.

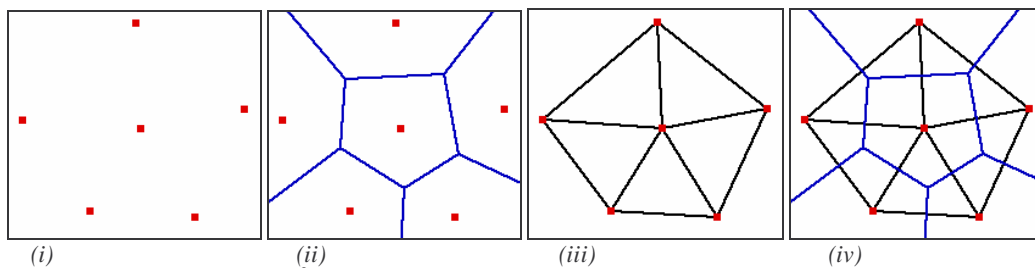


Figure 2.1 (i) 6 Nodes in  $\mathbf{R}^2$ . (ii) The Voronoi diagram. (iii) The Delaunay triangulation. (iv) both the Delaunay triangulation and the Voronoi diagram.

For the sake of argument, assume there exists five vectors in  $\mathbf{R}^2$  as depicted in figure 2.1(i) and that we refer to these vectors as nodes. The Voronoi diagram has the property that for each node every point in the region around that node is closer to that node than to any of the other nodes. The Delaunay triangulation is the graph where nodes with a common Voronoi edge are connected by an edge (see Fig. 2.1 (iii)). Alternately, it can be defined as a triangulation of the nodes with the additional property that for each triangle of the triangulation, the circumcircle of that triangle does not contain any other nodes. These two closely related data structures have been found to be among the most useful data structures of the field of Computational Geometry.

Now that these concepts have been introduced, we proceed with the introduction of the Growing Neural Gas (GNG) algorithm [4]. The GNG algorithm, published by Bernd Fritzke, is an unsupervised incremental clustering algorithm. Given some input distribution in  $\mathbf{R}^n$ , GNG incrementally creates a graph, or network of nodes, where each node in the graph has a position in  $\mathbf{R}^n$ . GNG can be used for vector quantization by finding the code-vectors in clusters. In GNG these code-vectors are represented by the reference vectors (the position) of the GNG-nodes. It can also be used for finding topological structures that closely reflects the

structure of the input distribution. GNG is an adaptive algorithm in the sense that if the input distribution slowly changes over time, GNG is able to adapt, that is to move the nodes so as to cover the new distribution.

Starting with two nodes the algorithm constructs a graph in which nodes are considered neighbours if they are connected by an edge. The neighbour information is maintained throughout execution by a variant of competitive Hebbian learning (CHL) [5], that is,

*For each input signal  $\bar{x}$  an edge is inserted between the two closest nodes, measured in Euclidian distance.*

The graph generated by CHL is called the “induced Delaunay triangulation” and is a sub-graph of the Delaunay triangulation corresponding to the set of nodes. The induced Delaunay triangulation optimally preserves topology in a very general sense [5]. CHL is an essential component of the GNG algorithm since it is used to direct the local adaptation of nodes and insertion of new nodes.

GNG only uses parameters that are constant in time. Further, it is not necessary to decide on the number of nodes to use a priori since nodes are added incrementally during execution. Insertion of new nodes ceases when a user defined performance criteria is met or alternatively if a maximum network size has been reached.

## **2.2 Growing Neural Gas**

The GNG algorithm assumes that each node  $k$  consists of the following:

- $\bar{w}_k$  - a reference vector, in  $\mathbf{R}^n$ .
- $error_k$  - a local accumulated error variable.
- A set of edges defining the topological neighbours of node  $k$ .

The reference vector may be interpreted as the position of a node in input space. The local accumulated error is a statistical measure that is used for determining appropriate insertion points for new nodes. Further, each edge has an age variable used to decide when to remove old edges in order to keep the topology updated. This is necessary since the nodes are moved; an action that might affect how CHL would generate the topology.

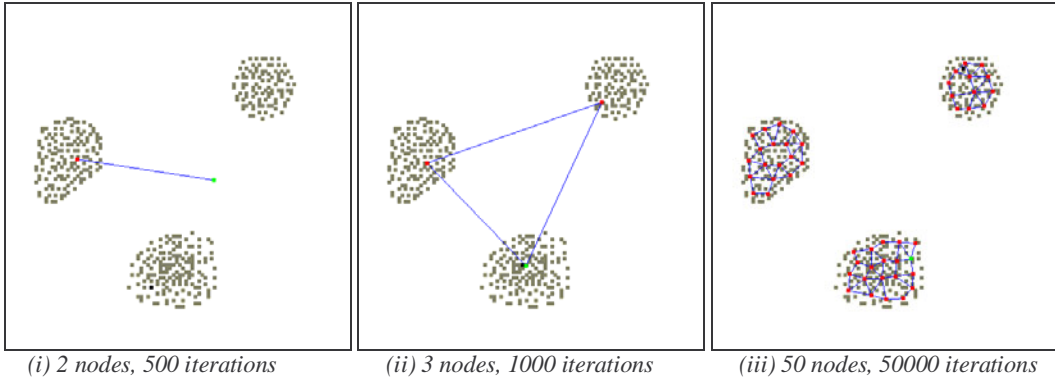


Figure 2.2. An illustration of the GNG algorithm. (i) The state of the GNG algorithm after 500 iterations, one node is located in the left most data cluster, the other node is oscillating between the top most and bottom most data clusters. (ii) After 1000 iterations, a third node has been inserted and the nodes now cover the three data clusters. (iii) After 50000 iterations, 50 nodes are spread out over the three data clusters matching the topology.

### 2.2.1 GNG Pseudo-Code

**INIT:** Create two randomly positioned nodes, connect them with a zero age edge and set their errors to 0.

- Generate an input vector  $\bar{x}$  conforming to some distribution.
- Locate the two nodes  $s$  and  $t$  nearest to  $\bar{x}$ , that is, the two nodes with reference vectors  $\bar{w}_s$  and  $\bar{w}_t$  such that  $\|\bar{w}_s - \bar{x}\|^2$  is the smallest value and  $\|\bar{w}_t - \bar{x}\|^2$  is the second smallest, for all nodes  $k$ .
- The winner-node  $s$  must update its local error variable so we add the squared distance between  $\bar{w}_s$  and  $\bar{x}$ , to  $error_s$

$$error_s \leftarrow error_s + \|\bar{w}_s - \bar{x}\|^2 \quad (1)$$

- Move  $s$  and its topological neighbours (i.e. all nodes connected to  $s$  by an edge) towards  $\bar{x}$  by fractions  $e_w$  and  $e_n$  of the distance.  $e_w, e_n \in [0,1]$

$$\bar{w}_s \leftarrow \bar{w}_s + e_w (\bar{x} - \bar{w}_s) \quad (2.1)$$

$$\bar{w}_n \leftarrow \bar{w}_n + e_n (\bar{x} - \bar{w}_n), \forall n \in \text{Neighbour}(s) \quad (2.2)$$

- Increment the age of all edges from node  $s$  to its topological neighbours. (3.1)
- If  $s$  and  $t$  are connected by an edge, then set the age of that edge to 0. If they are not connected then create an edge between them. (3.2)



- If there are any edges with an age larger than  $a_{max}$  then remove them. If, after this, there are nodes with no edges then remove these nodes. (3.3)

- If the current iteration is an integer multiple of  $\lambda$  and the maximum node-count has not been reached, then insert a new node. Insertion of a new node  $r$  is done as follows:

- Find the node  $u$  with largest error.
- Among the neighbours of  $u$ , find the node  $v$  with the largest error.
- Insert the new node  $r$  between  $u$  and  $v$  as follows:

$$\bar{w}_r \leftarrow \frac{(\bar{w}_u + \bar{w}_v)}{2} \quad (4)$$

- Create edges between  $u$  and  $r$ , and  $v$  and  $r$ , and then remove the edge between  $u$  and  $v$ .
- Decrease the error-variables of  $u$  and  $v$  and set the error of node  $r$ .

$$error_u \leftarrow \alpha \times error_u \quad (5.1)$$

$$error_v \leftarrow \alpha \times error_v \quad (5.2)$$

$$error_r \leftarrow error_u \quad (5.3)$$

- Decrease all error-variables of all nodes  $j$  by a factor  $\beta$ .

$$error_j \leftarrow error_j - \beta \times error_j \quad (6)$$

- If the stopping criterion is not met then repeat. The criterion might be for example the performance on a test set is good enough, or a maximum number of nodes has been reached, etc.

## 2.3 GNG Explained

### 2.3.1 The Local Accumulated Error

In (1) the local error is updated. Updating the error with the squared distance to the input is a way of detecting nodes that cover a larger portion of the input distribution. The local error is a statistical measure and nodes that cover a larger portion of the input distribution will have a faster growing error than other nodes, statistically. Large coverage is equivalent to larger updates of the local error, since inputs at greater distances will be mapped to the node. Since we want to minimize the errors, knowing where the error is large is useful when inserting new nodes. In (6) a global decrease of all local errors is performed, the reason for this is to give recent errors greater influence and to keep the local errors from growing out of proportion.

### 2.3.2 Node Movements

Equations (2.1) and (2.2) deal with node movements, or the adaptation of centres. The principle is the same for both the winner node and its neighbours. Figure 2.3 shows the idea behind the movement of the winner-node.

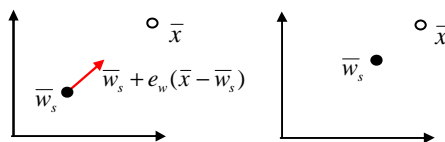


Figure. 2.3, Winner node movement

The winner-node  $s$  is translated along the scaled difference vector  $(\bar{x} - \bar{w}_s)$ . The scaling amount is denoted  $e_w$  and is a value between 0 and 1. The movement is linear, and the further the node is from the input the greater distance it will be translated. This is also true for the neighbour movement. However, the neighbour translation vector is normally scaled with a constant ( $e_n$ ) much smaller than  $e_w$ , which results in a smoother behaviour. The  $e_w$  and  $e_n$  parameters should not be set too high as this will result in a very unstable graph where the nodes move around too much. Setting them too low will make the adaptation slow and ineffective. Experimenting with these values has led to the following boundary-values, values greater than  $e_w = 0.3$  are considered big and  $e_w = 0.05$  is a normal value, meaning it performs well under most conditions. A normal value for  $e_n$  is 0.0006 or at preferably one to two orders of magnitude smaller than  $e_w$ . However, these values vary with the input distribution and the other parameters, see the experiments in section 2.4.

### 2.3.3 Edges and the Induced Delaunay Triangulation

Since the nodes are slowly being moved around, they may cause the current construction of the Delaunay triangulation to become invalid. The Delaunay triangulation is a slowly moving target as a direct result of the nodes being moved. A local aging process is used to invalidate edges that are not part of the Delaunay triangulation. This is done in (3.1). In (3.2) we create or update the edges in a CHL fashion. Edges that are updated have their age reset to 0, this is done to indicate that those edges still belong the current Delaunay triangulation and thus preventing them from being removed for now. In (3.2) we remove edges that should not be included in the Delaunay triangulation. They are identified by the fact that they are too old, which means CHL have not detected any correlated activity in  $a_{max}$  steps. If the removal of an edge results in a node with no edges, then it is a so-called *dead node* and it is safe to remove it.

### 2.3.4 Node Insertion

In GNG, nodes are inserted in a fixed interval manner. Every  $\lambda^{\text{th}}$  iteration a new node is inserted between the node with the largest error and its neighbour with the largest error. Having a fixed insertion rate policy might not always be desirable,

since it may lead to unnecessary or untimely insertions. A desirable insertion policy might perhaps be based directly on the local errors or a global mean error. For instance, a threshold constant could be used such that when the mean squared error is larger than the threshold, a new node is inserted. Another alternative is a combination of fixed insertion and error based insertion.

In the case of fixed insertion rate, the  $\lambda$  parameter has significant impact on the performance of the algorithm. Setting it too low will result in poor initial distribution of nodes, since the statistical local error will be badly approximated and since the nodes have not had a chance to distribute themselves over the input space. One consequence of using a very low  $\lambda$  is that the mean error will decrease rapidly in the beginning. But, in the long run it will take many more iterations to reach the same low mean error we would reach with a “normal”  $\lambda$  value. The reason is that the nodes must adapt for a longer period to allow them to cover the input distribution better, since they were placed without the proper statistics.

A too low value of  $\lambda$  also increases the risk of nodes becoming *inactive nodes*, meaning nodes that will not adapt any more since they are not close enough to the inputs. Inactive nodes are a waste of resources, which also implies that in the above scenario we might never reach the same low mean error since we do not have enough active nodes. Setting the  $\lambda$  parameter too high, on the other hand, will result in slow growth and requires the algorithm to run for many iterations, however nodes will be well distributed.

When inserting a new node, (4) specifies that it should receive the interpolated position of the node  $u$  with the largest accumulated error and the neighbouring node  $v$  with the largest accumulated error. Since we want to minimize the error, placing the new node at the median position will naturally decrease the Voronoi regions (the coverage) of both  $u$  and  $v$  and thus contribute to the minimisation of their future errors.

The subsequent decrease of the error variables of nodes  $u$  and  $v$ , (5.1) and (5.2), is motivated by the argument that when the new node has been inserted the current local errors of  $u$  and  $v$  are invalidated. This makes sense as the new node has assumed some of the coverage of the input distribution from both  $u$  and  $v$  and since the error in essence represents the coverage of the node (see 2.3.1), it is now invalid. Another reason for decreasing the errors is to prevent the next node insertion from occurring in the same region. If more nodes need to be inserted in this region, the errors will reflect this eventually. However, to what extent it should be decreased is not easy to say and most likely depends on the other parameters as well as the input distribution.

The particular initialisation-value of the error of the new node in (5.3) is not easy to motivate mathematically. It is given the newly decreased error of node  $u$  [4],

$$error_u \leftarrow \alpha \times error_u .$$

Initialising the error in this fashion is a way of approximating the error of the new node, had it been present as long as its “parent”. Further, it seems the importance of the details regarding the initialisation of the new error are not to be over-emphasized. In the DemoGNG v1.5 implementation [6] and in SGNG [7] the new error is initialised as the mean of  $error_u$  and  $error_v$ .

## 2.4 Experiments with GNG

Experiments will be conducted with the purpose is of illustrating some of the aspects of the GNG algorithm. All experiments are performed using the implementation described in section 6, and specifically the (5.3) error initialisation was chosen to be the same as in DemoGNG v1.5 [6]. In the following experiments all GNG mean-errors are computed every 500<sup>th</sup> iteration, as the mean of the 500 latest errors (squared distances) between inputs and winner-nodes.

### A Discrete Distribution

Figure 2.4.1, illustrates how GNG successfully finds the topological structure of a discrete stationary input distribution. As can be observed the mean error decreases, as the nodes are adapted to the input distribution. The mean error, displayed in figure 2.4.2 reached a minimum that was approximately 0.0022.

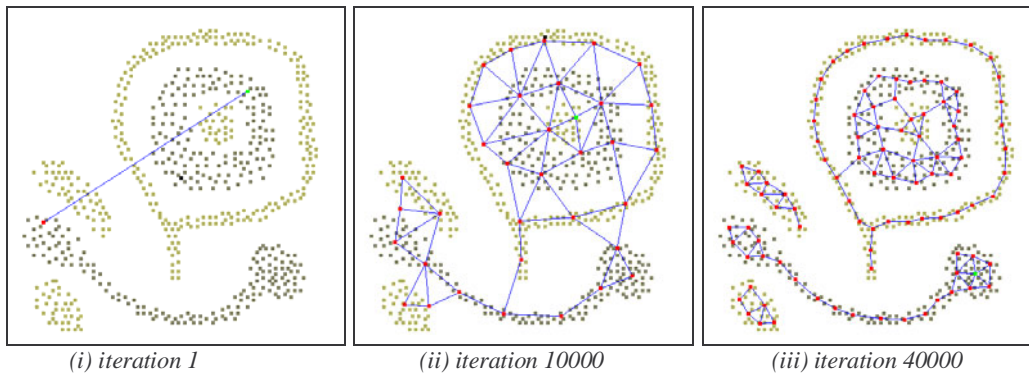


Figure 2.4.1. Three states of GNG using a discrete input distribution of 613 two-dimensional points between  $(-1,-1) \dots (1,1)$  (i) is GNG at the start state, (ii) is after 10000 iterations, with 35 nodes and (iii) is after 40000 iterations, with a total of 100 nodes.

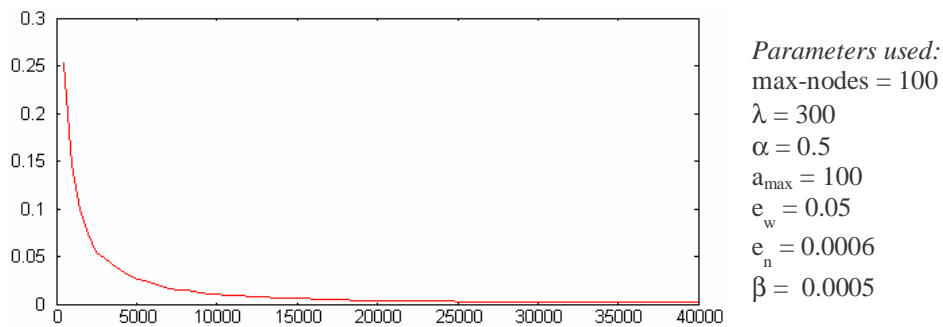


Figure 2.4.2. The mean local-error.

### A Jumping Distribution

Figure 2.4.3 illustrates the inactive-node scenario that occurs with rapidly moving distributions. Most of the nodes are positioned during the first 19999 iterations. When the distribution jumps (at 20000 iterations) almost all nodes are left behind and become a *inactive nodes*. A few nodes are inserted after that point but most of the resources of the network are wasted, given the distribution does not revisit any region close to these nodes in the future. The lowest mean reached was approximately 0.00247, and after the jump approximately 0.0143. Figure 2.4.4 represents the mean error.

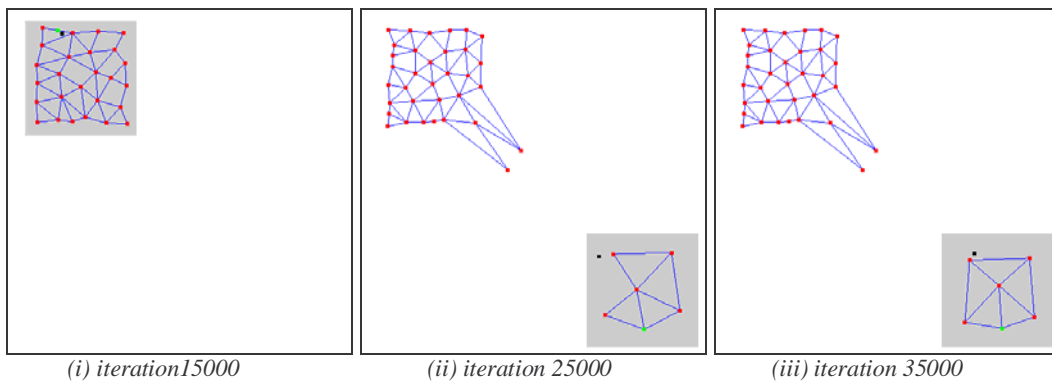


Figure 2.4.3 Three states of GNG using a non-stationary input distribution of two-dimensional points equally distributed, first in the ranges  $(-1, 0.65) \dots (-0.65, 1)$  and after 20000 iterations  $(0.65, -1) \dots (1, -0.65)$ . (i), The state after 15000 iterations, with 27 nodes. (ii), The state after 25000 iterations, with 40 nodes. (iii), The state after 35000 iterations, with 40 nodes.

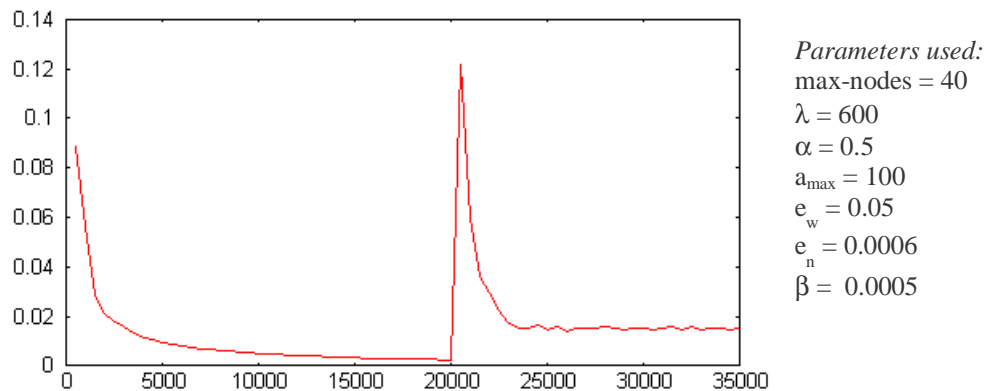


Figure 2.4.4 The mean local-error. Notice how the error spikes at iteration 20000, which is where the distribution jumped. The following new minimum is due to the 34 inactive nodes not contributing anything, and 6 nodes is not enough to lower the error to the same level as before.

### A Slowly Moving Distribution

Figure 2.4.5 illustrates the ability of GNG to track a slowly moving distribution. Due to the slow change in the distribution, all 30 nodes can be adapted sufficiently to avoid being left behind. The net achieves a lowest mean oscillating slightly around 0.00155, figure 2.4.6 is the mean error.

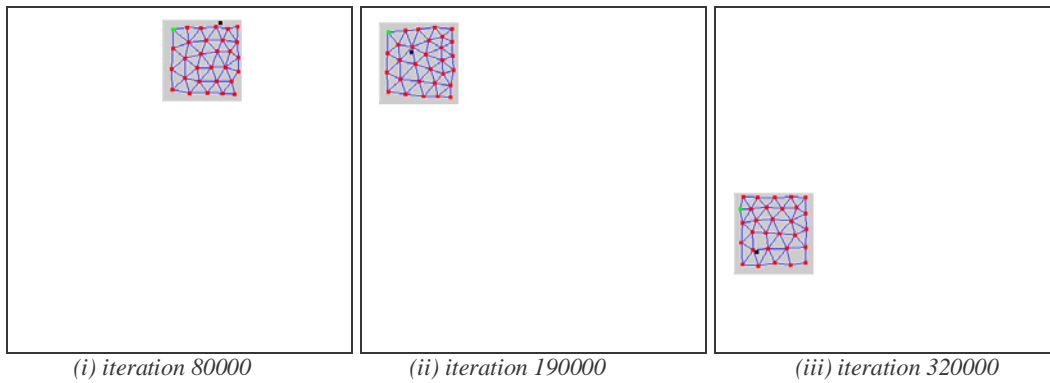


Figure 2.4.5 Three states of GNG tracking a non-stationary, slowly moving, input distribution of two-dimensional points equally distributed in the range  $(-0.25, -0.25) \dots (0.25, 0.25)$  and then translated so as to move slowly along the inside of the edges of the square  $(-1, -1)$  to  $(1, 1)$ . The centre of the distribution is moved by  $0.00000, 8$  in the described manner, each iteration. (i), The state after 80000 iterations, with 30 nodes. (ii), The state after 190000 iterations, with 30 nodes. (iii), The state after 320000 iterations, with 30 nodes.

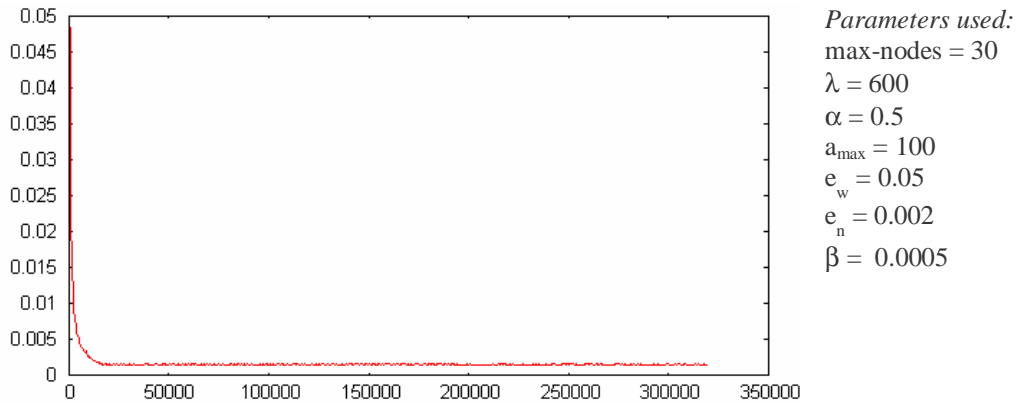


Figure 2.4.6 The mean local-error, notice that although the error oscillates slightly it is still quite low. The net reached its max node-count at iteration 16800, and after that, the mean error is oscillating slightly around 0.00155.

### Fixed Insertion-Rate Problems

Figure 2.4.7 illustrates the problem caused by using too low a value for the  $\lambda$  parameter. The discrete distribution from figure 2.4.1 ( $\lambda=300$ ) was chosen and the same parameters used, except for the  $\lambda$  parameter, which was set  $\lambda=5$ . It can be seen how the error decreased rapidly at first, but how it fails to reach the same low values as in 2.4.2. We can also see some inactive nodes that are unable to contribute anything since they are cut-off from the rest of the net and will never receive any inputs. These are the main problems that may occur when using a too low a value of  $\lambda$ . On the other hand, if we accept a slightly higher mean error, using a low  $\lambda$ , has the advantage of dropping quite fast to quite a low level, but the disadvantage of utilising unnecessarily many nodes for that error-level. 2.4.8 is the mean error graph. It reaches 0.0038 at about 12500 iterations using the maximum allowed 100 nodes, whereas 2.4.2 reaches that same error-level at about

21000 iterations using only 70 nodes. After that point however, the roles change, 2.4.8 reaches 0.0035 at about 25000 iterations using 100 nodes whereas 2.4.2 reaches the same level at about 22500 iterations using just 75 nodes.

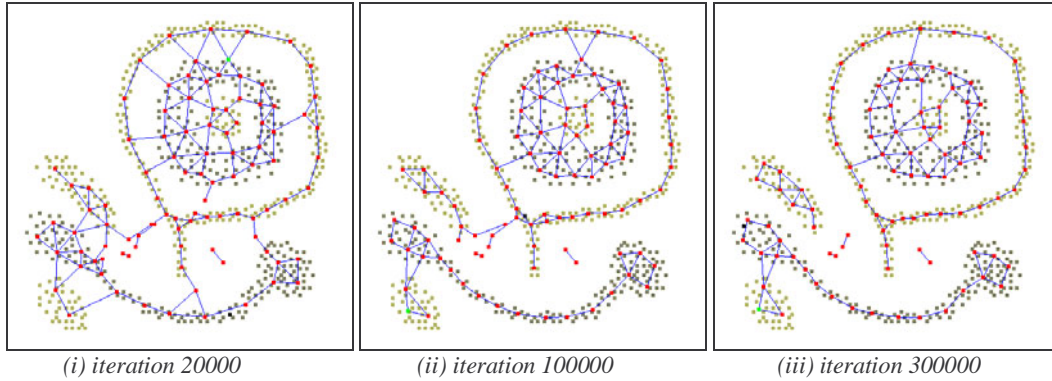


Figure 2.4.7, illustrates the insertion-rate problem. The same parameters and distribution as in figure 2.4.1 were used except for  $\lambda=5$ . (i), after 20000 iterations, we can see clearly that the nodes are distributed poorly, some are nodes are not covering the input distribution very well. (ii), after 100000 iterations, some nodes still have not adapted sufficiently. (iii), finally after 300000 iterations, all active nodes have adapted to the input distribution, unfortunately 6 inactive nodes exist as a result of the low  $\lambda$ .

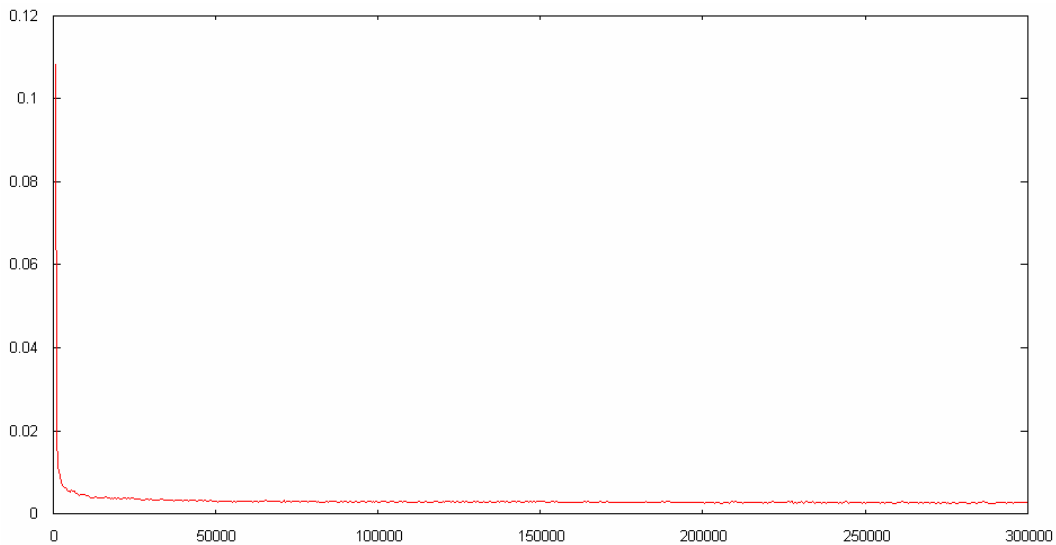


Figure 2.4.8, The mean error, although it decreases rapidly at first, it takes quite many iterations before it reached the similar low-level as in 2.4.2. After 10000 iterations the error is at approximately 0.00439, and in 2.4.2 it is 0.01066. After 40000 iterations it is approximately 0.00309, and in 2.4.2 it is 0.00212. After 100000 iterations the error is approximately 0.00277, and as can be seen on the graph the error does not decrease much after that, in fact it never decreases below 0.00245, and remains between 0.00245 and 0.00288. Parameters used:  $\max\text{-nodes} = 100$ ,  $\lambda = 5$ ,  $\alpha = 0.5$ ,  $a_{\max} = 100$ ,  $e_w = 0.05$ ,  $e_n = 0.0006$ ,  $\beta = 0.0005$ .



## ***2.5 Summary and Discussion***

GNG only uses parameters that are constant in time. Further, because of the incremental nature of GNG, it is not necessary to decide on the number of nodes to use a priori. This is an important feature compared to, for example, the k-means clustering algorithm, since several trials may be required to determine an appropriate number of centres to use. With GNG, insertion of new nodes continues until some user defined performance criteria are met or alternatively if a maximum network size has been reached.

The properties mentioned above make GNG an attractive candidate for problems where we know nothing or little about the input distribution, cases where deciding on network size and decaying parameters is very difficult or impossible.

The GNG algorithm is best suited to handle stationary distributions or slowly moving distributions. Applying GNG to a rapidly changing distribution with no repetition is very wasteful, in the sense of number of nodes needed, since most nodes will be left behind and will rarely or never contribute.

There are variations to the GNG algorithm not covered in this report that might be interesting to try. For example, standard GNG is based on linear movements of nodes, but one might try using non-linear movements, even gravity-inspired movements. In the latter scenario, inputs would be analogous to gravity wells, pulling the nodes closer in accordance with the laws of gravity.

Another minor variation is to draw the positions of the first two nodes directly from the input distribution instead of randomising them. This probably has little effect in the long run, since the nodes will adapt. However it might affect the very early results.



## 3 GNG with Utility Factor

### 3.1 Introduction to GNG with Utility Factor

The desire for tracking non-stationary distributions has given rise to GNG with Utility factor (GNG-U), published by Bernd Fritzke [8]. As you can tell by the name, it is a derivative of the GNG algorithm, published by the same author, which we discussed in section two.

The greatest weakness of the GNG algorithm is its inability to adapt to rapidly changing distributions. The GNG-U algorithm on the other hand, is designed to handle these scenarios by relocating less useful nodes. In essence, what GNG-U does is remove nodes that contribute little to the reduction of error, in favour of inserting them where they would contribute more to the reduction of the error. However, a carelessly set utility-factor affects the behaviour of the algorithm.

### 3.2 GNG with Utility Factor

The difference between the GNG and the GNG-U algorithm is not great but its impact on ability/performance is significant.

GNG-U is based on the same assumptions as GNG with the addition of:

- For all nodes  $n$ , we include a local variable, the utility  $U_n$  of that node.

The modifications of the pseudo-code are minor.

- In GNG step 3 after the local error of node  $s$  has been updated we now add the update rule for the utility  $U_s$  of the winner-node  $s$ .

$$U_s \leftarrow U_s + error_t - error_s$$

- In GNG step 7 nodes that have no edges are deleted. In GNG-U however, the removal criterion is based on other factors. Remove node  $i$  with the smallest utility  $U_i$  if

$$\frac{error_j}{U_i} > k$$

where  $j$  is the node with the greatest error and  $k$  is a constant parameter. We remove node  $i$  if the utility falls below a certain fraction of the error.

- In GNG step 8 we add a new node. The utility of the new node  $r$  is initialised to the mean of  $U_u$  and  $U_v$ .<sup>1</sup>

$$U_r \leftarrow \frac{U_u + U_v}{2}$$

- In GNG step 9 the utility is decayed for all nodes in the same manner as the error and with the same decay constant.

$$U_k \leftarrow U_k - \beta \times U_k$$

### 3.3 GNG-U Explained

#### 3.3.1 The Utility Update Rule

The utility-update is the direct increase in squared error for an input signal  $\bar{x}$  if the winner-node  $s$  was non-existent. In this case, the input would be mapped to the second closest node  $t$  and the increase in error for that particular signal  $\bar{x}$  would be

$$\begin{aligned} U_s &\leftarrow U_s + \|i - w_t\|^2 - \|i - w_s\|^2 \\ &\Leftrightarrow \\ U_s &\leftarrow U_s + error_t - error_s \end{aligned}$$

The utility is updated each iteration for the winner-node  $s$ . This yields an approximation of how much a particular node  $j$  decreases the error of the input signals in its region. Removing a high-utility node, compared to a low-utility node, means the increase in error in that region will be greater. The utility of a node becomes small either when the neighbours are very close or when the node rarely or never wins (since the utility is decayed).

#### 3.3.2 Removal Criterion and Utility

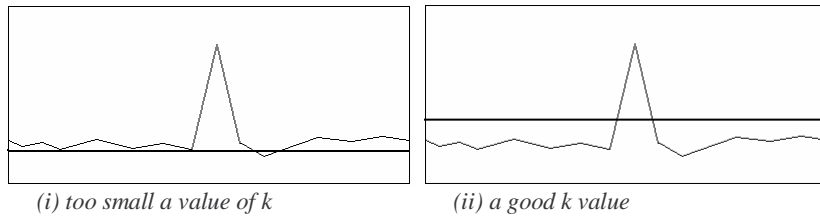
As described above the utility of a node approximates the increase in error from removing that node. Further, the expected decrease in error by inserting a new node near the node  $j$ , with the greatest error, is some fraction of  $error_j$ . To determine if the error can be reduced by relocating the node  $i$ , with the lowest utility  $U_i$ , to the vicinity of node  $j$  we check if  $U_i$  falls below a certain fraction of  $error_j$ . The constant  $k$  decides the minimum size of that fraction.

The constant  $k$  defines how sensitive the removal should be, a small  $k$  gives rise to frequent deletions and consequently fewer nodes are left behind. There is also the

---

<sup>1</sup> Note that the utility-variables of nodes  $u$  and  $v$  are not decreased in the same fashion as the error-variables are, in fact they are left perfectly intact. This is commented on in 3.3.3

risk of a low  $k$  keeping the total number of nodes very low since it might relocate nodes when  $error_j/U_i > k$  occurs, simply because  $k$  is too small. Explanation by example, assume we have a distribution where each input has equal probability. Assume the nodes are spread equally over the inputs. The ratio error/U will average some fairly stable value in this scenario. If  $k$  is sufficiently less than that value, nodes will be removed and relocated even though there was a “good” distribution of nodes, see figure 3.1(i). The reason being that  $k$  was too small. The purpose is to identify ratio-spikes, this purpose is subverted if  $k$  is too small.



*(i) too small a value of  $k$*                       *(ii) a good  $k$  value*  
 Figure 3.1 An illustration of the  $k$ -value problem. The horizontal line represents the  $k$  value and the jagged line is the error/U ratio. This is a simplification to facilitate explaining the phenomenon.

A large  $k$  on the other hand, causes less frequent deletions. In fact, the constant  $k$  can be seen as a measure of how much “memory” GNG-U should have.

### 3.3.3 Utility Initialisation for New Nodes

Fritzke does not define the initialisation of the utility variable for a new node. However, in the DemoGNG v1.5 implementation [6] it is defined as the mean of  $U_u$  and  $U_v$ . Note also that there is no mention of a decrease of the utilities of nodes  $u$  and  $v$  corresponding to the error decrease in GNG after a new node has been inserted. It stands to reason that the utilities of  $u$  and  $v$  should be decreased in the same manner as the errors. This is something to test in future implementations.

## 3.4 Experiments on GNG-U

The purpose of these experiments is to illustrate some of the aspects of the GNG-U algorithm. The mean errors are computed every 500<sup>th</sup> iteration, as the mean of the 500 latest errors (squared distances) between inputs and winner-nodes.

### Relocation of nodes

Figure 3.4.1 illustrates how GNG-U successfully relocates nodes in a jumping distribution scenario. Figure 3.4.2 is the mean error.

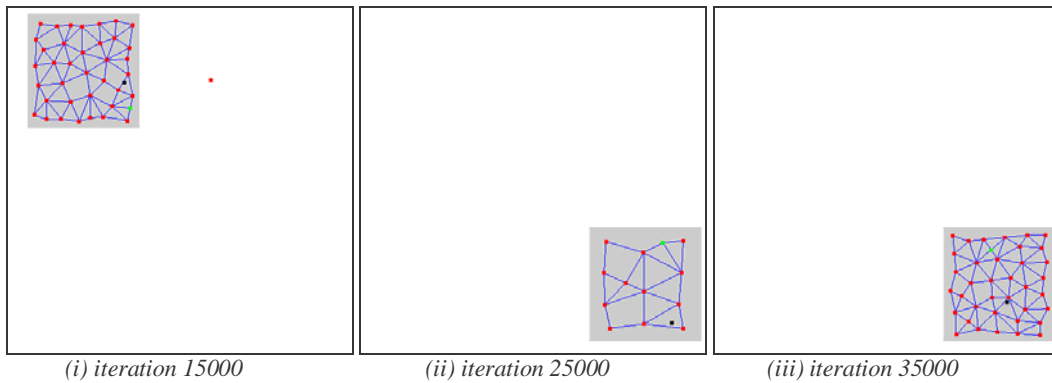


Figure 3.4.1 Three states of GNG using a non-stationary input distribution of two-dimensional points equally distributed, first in the ranges  $(-1,0.65)\dots(-0.65,1)$  and after 20000 iterations  $(0.65,-1)\dots(1,-0.65)$ . (i), The state after 15000 iterations, with 39 nodes. (ii), The state after 25000 iterations, with 13 nodes. (iii), the state after 35000 iterations, with 38 nodes.

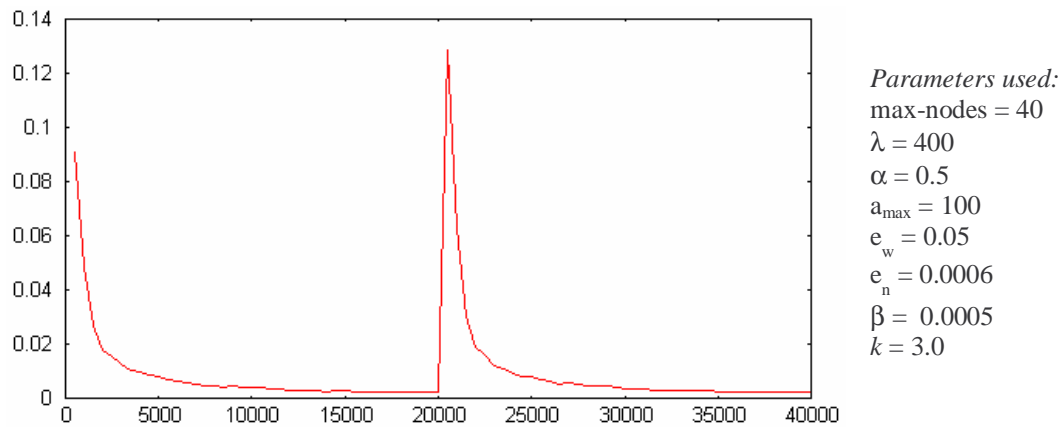


Figure 3.4.2 The mean error. At 20000, the distribution jumps and we consequently get an increase in mean error. As the GNG-U algorithm removes the inactive (useless) nodes they are available to be placed where the distribution is located after the jump and the mean error can be decreased to the same level as before the jump.

### The utility boundary $k$ (low and high)

To illustrate how the  $k$  parameter influences the behaviour of the GNG-U algorithm, a moving distribution has been chosen. In figure 3.4.3, a high value of the  $k$  parameter will be used to illustrate the “memory phenomenon” and in figure 3.4.5, a low value will be used to illustrate frequent deletions. The mean errors are found in 3.4.4 and 3.4.6 respectively.

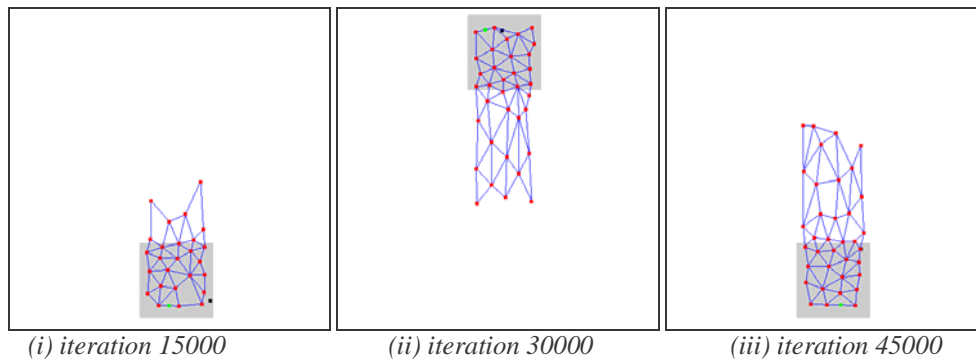


Figure 3.4.3, Three states of GNG-U tracking a non-stationary down-up moving, input distribution of two-dimensional points equally distributed in the range  $(-0.25, -0.25) \dots (0.25, 0.25)$  and then translated so as to move along a vertical path, down then up etc. The centre of the distribution is moved by 0.0001 in the described manner each iteration. (i), The state after 15000 iterations, with 27 nodes. (ii), The state after 30000 iterations, with 36 nodes. (iii), The state after 45000 iterations, with 37 nodes. As can be seen, a trail of nodes are left dragging behind the distribution, this is a result of the high  $k$ -parameter value. The number of nodes vary between 27 and 52.

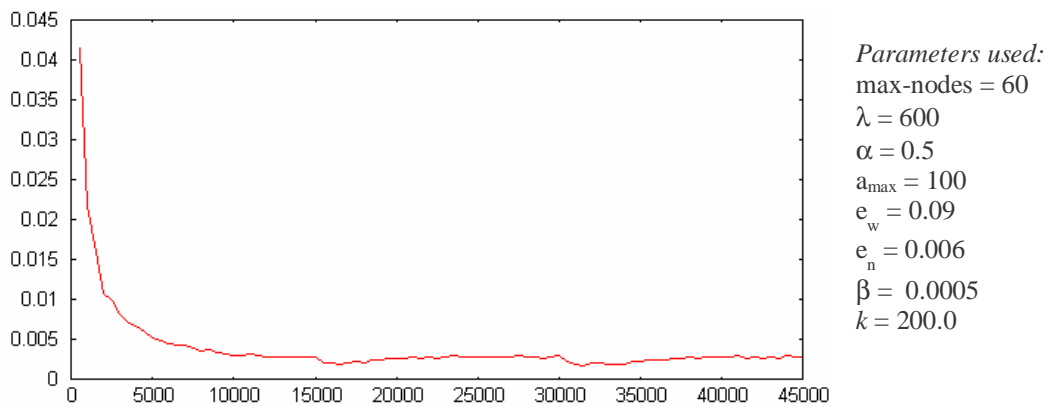


Figure 3.4.4. The mean error. The unevenness of the error depends on nodes being relocated and on the motion of the distribution.

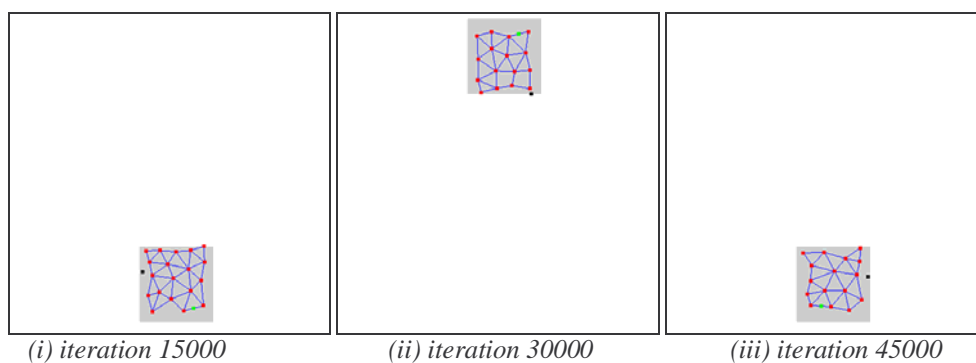


Figure 3.4.5. Three states of GNG-U tracking a non-stationary down-up moving, input distribution of two-dimensional points equally distributed in the range  $(-0.25, -0.25) \dots (0.25, 0.25)$  and then translated so as to move along a vertical path, down then up etc. The centre of the distribution is moved by 0.0001 in the described manner each iteration. (i), The state after 15000 iterations, with 20 nodes. (ii), The state after 30000 iterations, with 17 nodes. (iii), The state after 45000 iterations, with 17 nodes.

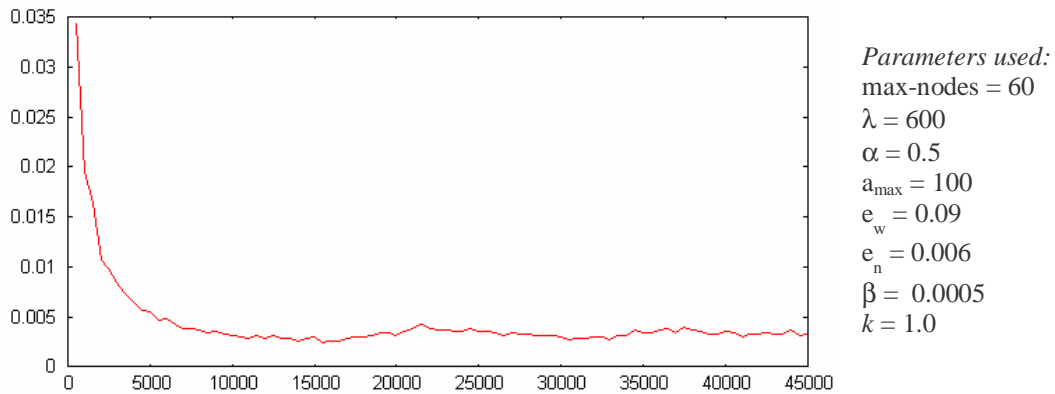


Figure 3.4.6 The mean error. The unevenness of the error depends on nodes being relocated and on the motion of the distribution. The error is slightly higher here than in 3.4.4, since fewer nodes are left in place due to the low value of the  $k$ -parameter. The number of nodes vary between 15 and 20.

### 3.5 Summary and Discussion

GNG-U is good at tracking rapidly moving distributions, and can successfully relocate nodes to avoid wasting resources. GNG-U can be used with stationary distributions as well, but that is not the purpose of the algorithm and GNG performs equally good or better in these respects. Depending on what behaviour is valued more, low resource-waste or memory, the  $k$ -parameter should be set accordingly. It determines the deletion-sensitivity and can have a limiting effect on the total number of nodes used.

Because of the similarities between GNG-U and GNG, the later parts of the discussion concerning GNG in section 2.5 also apply to GNG-U.

## 4 Radial Basis Function Networks

### 4.1 Introduction to RBF Networks

An Artificial Neural Network (ANN) can be described as a massively parallel system consisting of simple interconnected processing units [9]. There are many types of ANNs but we will only cover the basics of the Radial Basis Function (RBF) network.

The RBF network is a fully interconnected feed forward network with one hidden layer, see figure 4.1.

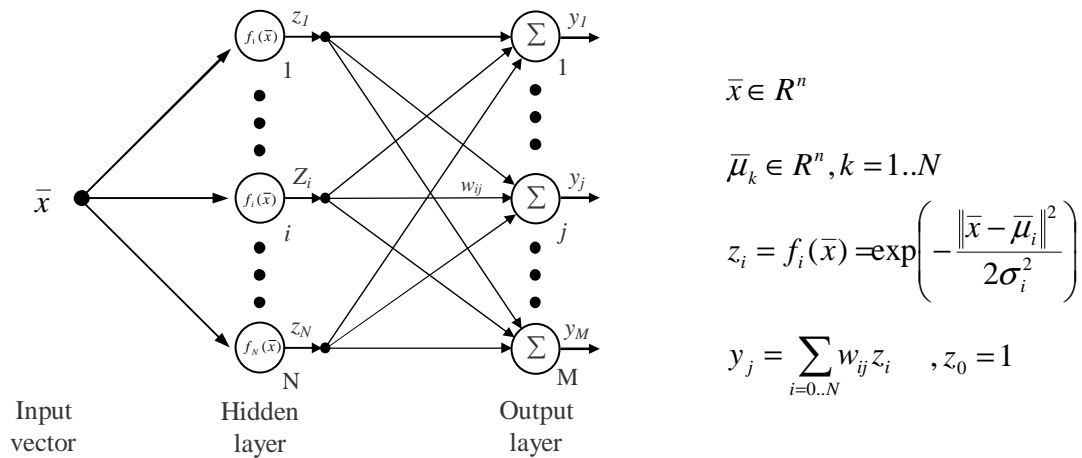


Figure 4.1, a Radial Basis Function Network with Gaussian hidden nodes.

Each node  $k$  in the hidden layer has a vector  $\bar{\mu}_k$  that determines the position of the node in input space and a standard deviation  $\sigma_k$  that defines the width of the local receptive field of node  $k$ . Each node in the hidden layer is connected to all other nodes in the output layer. Each connection has a weight,  $w_{ij}$ , which is a real value. The weights are updated during training and we refer to this adaptation as learning. The activation function of the hidden nodes, denoted  $f_i(\bar{x})$  in figure 4.1, can be any strictly positive radially symmetric function with a unique maximum at its centre  $\bar{\mu}_k$ , but we will assume the Gaussian function is used, since it is the common case.

When inputs are presented to the network, only the nodes that have the input within their receptive fields are activated. The level of activation depends on the distance to the input and decreases rapidly with increasing distance. The output layer consists of nodes with linear output functions. Further, each output node has a bias, which can be viewed as an extra weight  $w_{0k}$  with the constant input 1. RBF networks are used in classification scenarios as well as in function approximation.

## 4.2 Training

Since the two layers in a RBF network perform different tasks, it is reasonable to separate their training. The separation is possible because of the local receptive field nature of the hidden units [10].

There are three main questions to answer in connection with the training of the hidden layer. How many hidden nodes do we need, what should the widths be and where should the nodes be placed. There are several methods designed to accomplish this to various degrees, typically the classical k-means algorithm is used as the clustering algorithm for the hidden layer. However, we will focus on the application of a slightly modified version of GNG as the clustering algorithm for the hidden layer.

When the hidden units have been trained, all that remains is to train the output-layer weights, which is done using the delta rule [10].

$$w_{ij} \leftarrow w_{ij} + \eta(d_i - y_i)z_j$$

where  $d_i$  is the desired response,  $y_i$  is the actual response,  $\eta$  is the step-size and  $z_j$  is the output from hidden node  $j$  (see *figure 4.1*).



## 5 Supervised GNG

### 5.1 Introduction to Supervised GNG

Supervised GNG or SGNG is described by Bernd Fritzke [7] as a new algorithm for constructing RBF networks. The SGNG algorithm can be described as an RBF network with a slightly modified version of the GNG algorithm as the method for constructing and managing the hidden layer.

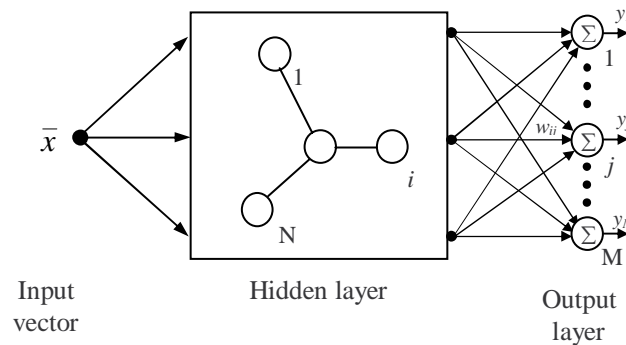


Figure 5.1 depicts a SGNG-network, it is a regular RBF network with a modified version of the GNG algorithm as the clustering algorithm for the hidden layer.

### 5.2 Supervised GNG

#### 5.2.1 Brief Description

For the sake of argument, we will discuss only classification and not function approximation, the general discussion that follows however, applies equally to both cases and when this is not the case, it will be pointed out.

Assume we want to classify  $n$ -dimensional vectors into  $M$  classes. Assume further that we code our outputs in a one-out-of- $M$  fashion, this means we will have one output node for each class, and that the output-node with the greatest value is the only one considered in each response from the RBF network. Training is done by presenting pairs of input and expected output vectors.

As in GNG, we start with two randomly positioned nodes that are connected by a neighbour edge. Note that the edge has no weight, since it is not part of the actual RBF network, but represents the fact that two nodes are neighbours, as in GNG. The neighbour information is maintained in the same manner as in GNG, by application of competitive Hebbian learning (CHL). The adaptation of the hidden nodes is also performed in the same manner as in the GNG algorithm, the winner node  $s$  is moved some fraction of the squared distance to the input, and the neighbours of  $s$  are moved an even smaller fraction of their squared distance to the input. The two most notable differences, the local error updates and the insertion criteria are discussed later.

## 5.2.2 SGNG Pseudo-Code

Let the input vector be denoted  $\bar{x} = (x_1, \dots, x_n)$ .

Let the desired output vector be denoted  $\bar{d} = (d_1, \dots, d_M)$ .

Let the output vector of the output-layer be denoted  $\bar{y} = (y_1, \dots, y_M)$ .

Let  $\sigma_j$  be the standard deviation, or width, of the RBF node  $j$ .

Let  $z_i$  be the output from the hidden node  $i$ , and  $w_{ij}$  the weight from hidden node  $i$  to output node  $j$ .

Let  $\eta$  be the step-size.

All weights are randomly initialised in the interval  $[-1, 1]$ .

**INIT:** Create two randomly positioned hidden-nodes, connected by an edge with age 0 and set their errors to 0. Randomly initialise the weights between these nodes and the output-nodes in the interval  $[-1, 1]$ .

- Generate an input vector  $\bar{x}$  conforming to some distribution, also let  $\bar{d}$  be the corresponding desired output vector.
- Locate the two nodes  $s$  and  $t$  nearest to  $\bar{x}$ , that is, the two nodes with reference vectors  $\bar{w}_s$  and  $\bar{w}_t$  such that  $\|\bar{w}_s - \bar{x}\|^2$  is the smallest value and  $\|\bar{w}_t - \bar{x}\|^2$  is the second smallest, for all nodes.
- Evaluate the net using the input vector  $\bar{x}$ . Adjust the output-layer weights by applying the delta rule.

$$w_{ij} \leftarrow w_{ij} + \eta(d_j - y_j)z_i$$

- The winning node  $s$  must update its local error variable so we add the squared error of the output

$$error_s \leftarrow error_s + \|\bar{d} - \bar{y}\|^2 \quad (1)$$

- Move  $s$  and its topological neighbours (i.e. all nodes connected to  $s$  by an edge) towards  $\bar{x}$  by fractions  $e_w$  and  $e_n$  of the distance.  $e_w, e_n \in [0, 1]$

$$\bar{w}_s \leftarrow \bar{w}_s + e_w(\bar{x} - \bar{w}_s)$$

$$\bar{w}_i \leftarrow \bar{w}_i + e_n(\bar{x} - \bar{w}_i), \forall i \in \text{Neighbour}(s)$$

- For each node  $j$  that was just moved, set the width of the RBF to the mean of the distances between  $j$  and all the neighbours of  $j$ .

$$\sigma_j = \text{mean}_n \left( \left\| \bar{w}_j - \bar{w}_n \right\| \right), \quad (2)$$

$$\forall n \in \text{Neighbour}(j)$$

- Increment the age of all edges from node  $s$  to its topological neighbours.
- If  $s$  and  $t$  are connected by an edge, then set the age of that edge to 0. If they are not connected then create an edge of age 0 between them.
- If there are any edges with an age larger than  $a_{max}$  then remove them. If, after this, there are nodes with no edges then remove these nodes. Recalculate the mean RBF widths of the affected nodes.
- If some *insertion criteria* is met then insertion of a new node  $r$  is done as follows: (3)
  - Find the node  $u$  with largest error.
  - Among the neighbours of  $u$ , find the node  $v$  with the largest error.
  - Insert the new node  $r$  between  $u$  and  $v$  as follows:

$$\bar{w}_r = \frac{(\bar{w}_u + \bar{w}_v)}{2}$$

- Create edges between  $u$  and  $r$ , and  $v$  and  $r$ , and then remove the edge between  $u$  and  $v$ .
- Recalculate the mean RBF widths of  $u$ ,  $v$  and  $r$ .
- Decrease the error-variables of  $u$  and  $v$  and set the error of node  $r$ .

$$\begin{aligned} \text{error}_u &\leftarrow \text{error}_u \times 0.5 \\ \text{error}_v &\leftarrow \text{error}_v \times 0.5 \\ \text{error}_r &\leftarrow (\text{error}_u + \text{error}_v) \times 0.5 \end{aligned}$$

- Initialise the weights, randomly in the interval  $[-1,1]$ , from node  $r$  to all nodes in the output layer.
- Decrease all error-variables of all nodes  $j$  by a factor  $\beta$ . This gives recently measured errors greater influence than older ones.

$$\text{error}_j \leftarrow \text{error}_j - \beta \times \text{error}_j$$

- If the stopping criterion is not met then repeat the process. (4)

## 5.3 SGNG Explained

### 5.3.1 The Local Accumulated Error

In SGNG, the local error is updated differently from in GNG. The local error of the winner node  $s$  is updated, in (1), with the squared output error of the RBF network, in other words the squared difference in real output and desired output.

By observing the local accumulated squared errors, we can identify nodes that exist in regions of input space where many misclassifications occur. It is logical to assume that nodes with large local accumulated errors received patterns from different classes since if they did not, the delta rule, with a properly set  $\eta$ , would have successfully modified the output layer weights and the local accumulated errors would not have grown so high. In function approximation, the error consequently represents the need for more gaussians to allow better approximation of that particular function-interval. With this argument in mind, it would seem sensible to use the local error information when deciding possible locations of new nodes. Another approach could be to use gradient information to distribute the error amongst the hidden nodes. This will be discussed in section 5.5.

### 5.3.2 The Mean Distance and RBF Width

As stated in the SGNG definition in [7] the widths of the RBF-nodes are set to the mean distance to their neighbours, a question that arises when reading this is how often the widths should be updated. Since this is not specified, a certain amount of freedom is given in this respect. In (2) we have chosen to update the widths of all nodes that have been affected each iteration. This is however not very economical in terms of execution time. Another approach is to update the widths every  $n$ th iteration, thus being able to control how closely we want to approximate the real mean widths, or one could use a discounting factor to allow the mean width to slowly adjust to its new values. However, such details will not be discussed in this report.

### 5.3.3 Insertion of New Nodes

The criteria for inserting new nodes, (3), could very well be a fixed insertion policy however a better method would be to observe the mean squared error per pattern or some independent validation set. If the squared error stops decreasing that can be interpreted as the delta rule and node movements not being able to adjust sufficiently to lower the error, this in turn means that for the current network size this is as good as it gets and that it is time to insert a new node. Of course, it could also be that the problem cannot be solved and that this is the reason for the error not dropping further.

The newly inserted node  $r$  receives the interpolated position of its “parents” and the parents errors are reduced by 50%, this reduction was decided heuristically

since it is difficult to motivate theoretically [7]. The purpose is the same as in the GNG algorithm, to prevent the next insertion to occur in the same place. If more nodes need to be inserted in this region, the errors will reflect this eventually.

### **5.3.4 Stopping Criteria**

The criteria for stopping, (4), could be defined as a maximum size that the network may reach. However, this is as difficult to do as for other RBF methods, because it requires knowledge about the distribution that we simply might not have. Another method is available to us because of the incremental nature of the algorithm. We can simply define a maximum error allowed, and train until the criterion is met. Yet, another approach that applies only to classification and not function approximation would be to have an upper limit on the allowed number of misclassifications. This is more practical in the classification case since the number of misclassifications converges towards lower values much faster than the error.

### **5.3.5 General Notes and Observations**

The constant adjusting of widths continuously disrupts some of the training of the output-layer weights, and so does the insertions of new nodes. The disruption caused by the insertion of a new node is in no way surprising since additional input is added to the summation in the output nodes. In addition, when a new node is inserted, new random-valued weights are created to link the new node to the output layer. The new weights introduce unpredictable and almost certainly erroneous contributions before they are trained.

Using SGNG in function approximation requires a larger number of iterations, since we want to achieve low errors, something not as essential in classification since we normally use decision thresholds to quantize the answers. Another point worth mentioning regarding function approximation, specifically interpolation of discretely sampled functions, is that the number of nodes used should be less than the number of samples. Since, otherwise we risk over-training the SGNG-net, it will start inventing erroneous values in between the samples that do not coincide with the real function-values we are trying to approximate.

## **5.4 Experiments**

Three experiments will be presented to illustrate the ability of SGNG to handle some typical function approximation and classification tasks.

### **The XOR function**

One of the standard benchmarks, the XOR problem in figure 5.4.1, can be solved by a SGNG net with one linear output. But, because of the adaptive nature of the algorithm, it is not practical to solve the XOR problem with just 2 nodes as they will be pulled back and forth between the four input ( (0,0), (1,0), (1,1), (0,1) ), each movement affecting the width of the RBFs. Using more than two nodes, for example 6, will introduce stability and reduce the lengths of the movements which in turn facilitates the training of the output layer weights. However, the XOR

function being a mapping  $\{(0,0), (1,0), (1,1), (0,1)\} \rightarrow \{0,1\}$  could equally well be, and is normally viewed as, a classification problem of two classes. In this experiment however, the function view was chosen since it illustrates the ability of SGNG to approximate a non-continuous function.

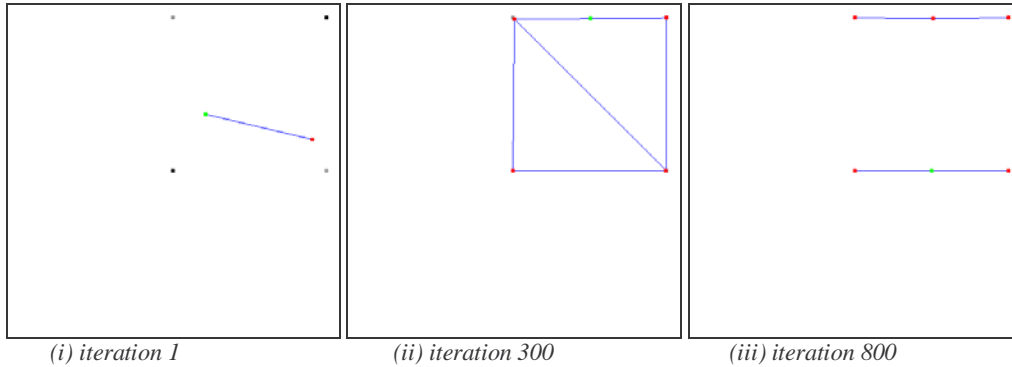


Figure 5.4.1 shows three states of the SGNG algorithm on the XOR problem. There are four inputs located at  $(1,1)$ ,  $(1,0)$ ,  $(0,1)$ ,  $(0,0)$ , each iteration one of them is selected at random with equal probability. (i), the initial iteration. (ii), after 300 iterations, all nodes but one have been inserted and moved into place. (iii), after 800 iterations we have reached the accepted mean squared-error threshold  $0.00001$ .

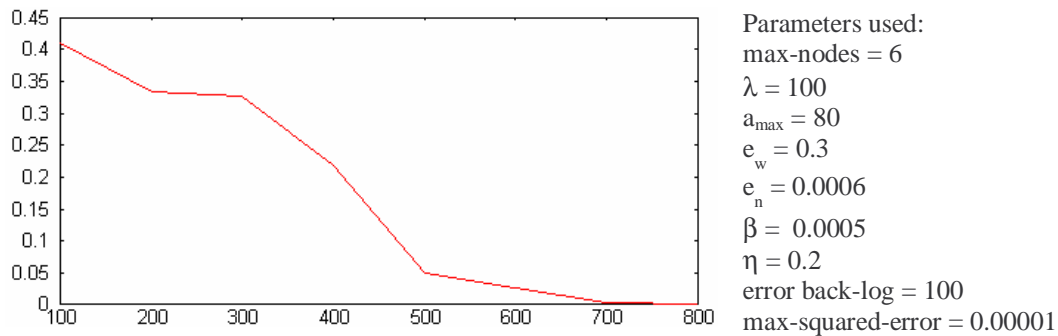


Figure 5.4.2 The mean error of the outputs of the SGNG net, calculated every  $100^{\text{th}}$  iteration. After 800 iterations, the mean squared error is less than  $0.00001$ . The mean error at 800 iterations is approximately  $5.83613e-06$ .

### Two classes in a discrete distribution

As an example of classification, we will use a discrete distribution with two classes depicted in figure 5.4.3. It is the same distribution as in Section 2.4, figure 2.4.1. The class-membership is denoted by the colour.

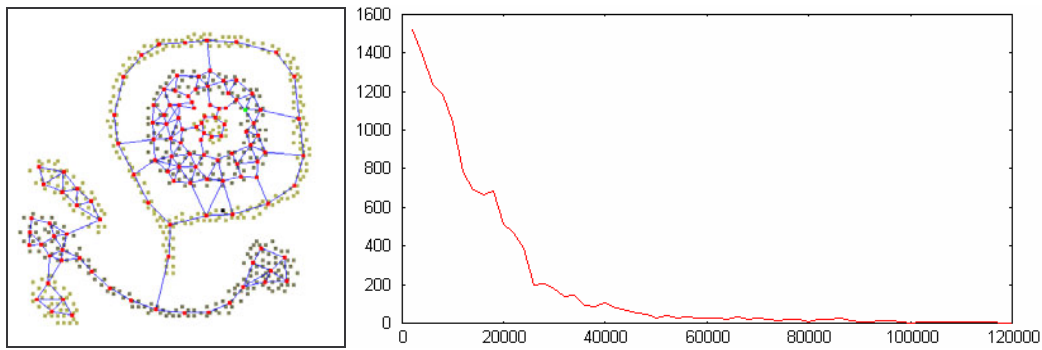


Figure 5.4.3, shows the discrete 2-classes 613-point distribution from section 2.4, and the misclassification graph from the training session. The SGNG-RBF net could successfully classify 2000 successive, randomly chosen, inputs after 12000 iterations using 120 nodes. The parameters used were:  $\text{max-nodes} = 120$ ,  $\lambda = 800$ ,  $a_{\text{max}} = 120$ ,  $e_w = 0.09$ ,  $e_n = 0.0004$ ,  $\beta = 0.02$ ,  $\eta = 0.05$ ,  $\text{misclassification-back-log} = 2000$ ,  $\text{allowed number of misclassification} = 0$ ,  $\text{decision difference minimum} = 0.0$ ,  $\text{decision min limit} = 0.65$ .

### Comparison between SGNG and GNG in a classification scenario

Having seen the SGNG performance in classifying the 613-point discrete distribution in figure 5.4.3, it is only natural to wonder how much the supervised error information really contributes to solving the classification problem. Figure 5.4.4 shows the standard GNG applied to classification of the same distribution as in 5.4.3 and using the same parameters. The difference in the placement of nodes can be seen clearly. In 5.4.4 (GNG) the nodes are more evenly distributed than in 5.4.3 (SGNG) where the density of nodes tend to increase in areas where it is difficult to classify the inputs.

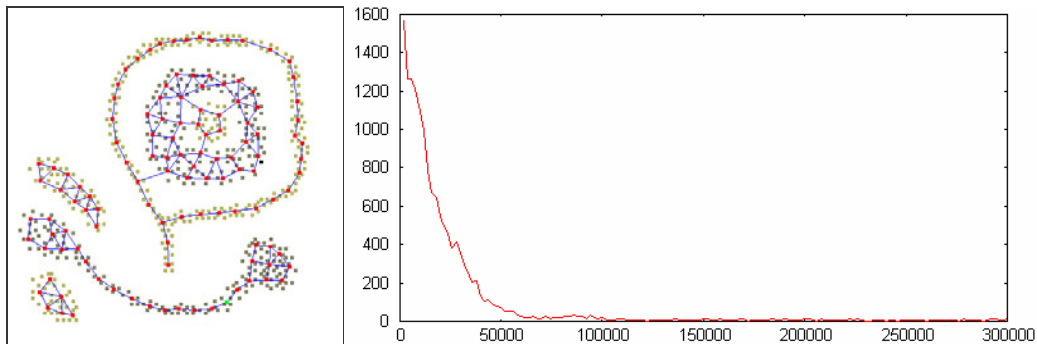


Figure 5.4.4, GNG (with RBF) shows the discrete 2-classes 613-point distribution from section 5.4.3, and the misclassification graph from the training session. The GNG-RBF net failed to classify 2000 successive, randomly chosen, inputs after 300000 iterations using 120 nodes. The exact same parameters as in 5.4.3 were used. The minimum number of misclassifications reached was 1, which is quite good, however at 120000 iterations the misclassification count was 9 and the misclassification count oscillated around that value until the experiment was terminated at 300000 iterations.

To further illustrate the difference in node-placement between GNG and SGNG and to illustrate a long-term issue with SGNG, consider the discrete distribution in

figure 5.4.5, which has a sharp, jagged class-border. The distribution is presented in pairs, SGNG and GNG, at three different iterations.

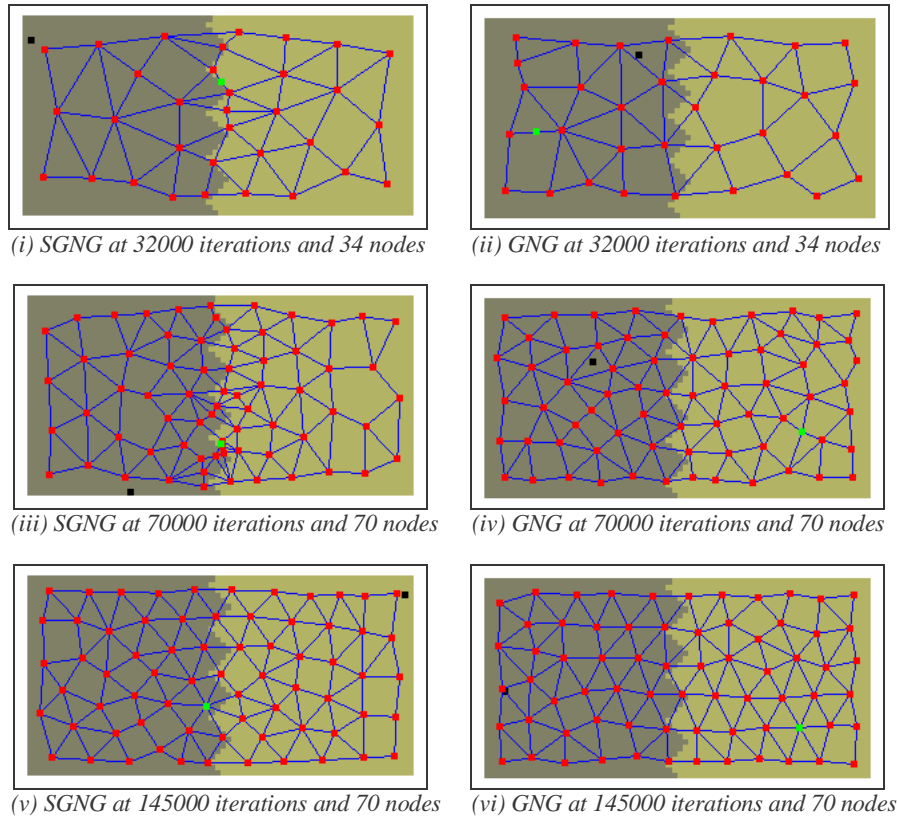


Figure 5.4.5. A uniform discrete distribution of 2520 points divided evenly into two classes. In each iteration, an input is randomly selected from the distribution in a uniform fashion. (i) and (ii) shows SGNG and GNG at 32000 iterations. (iii) and (iv) shows the SGNS and GNG at 70000 iterations. The maximum number of nodes is reached at 68000 iterations. (v) and (vi) shows the states after 145000 iterations, it can be seen that SGNG and GNG have produced very similar node-placements. The parameters used were:  $\text{max-nodes} = 70$ ,  $\lambda = 1000$ ,  $a_{\text{max}} = 60$ ,  $e_w = 0.04$ ,  $e_n = 0.0001$ ,  $\beta = 0.02$ ,  $\eta = 0.03$ ,  $\text{misclassification-back-log} = 5000$ ,  $\text{allowed number of misclassification} = 0$ ,  $\text{decision difference minimum} = 0.0$ ,  $\text{decision min limit} = 0.65$ .

After 32000 iterations we can see a distinct difference in the placement of nodes between 5.4.5(i) and 5.4.5(ii). In 5.4.5(i) SGNG places nodes near the jagged edge since that is where most of the misclassifications occur. In 5.4.5(ii), GNG places the nodes based on the local accumulated errors, which results in rather evenly placed nodes. These two different behaviours become even more announced at 5.4.5(iii) and 5.4.5(iv). The maximum number of nodes is reached at 68000 iterations at which point the supervised error-information ceases to affect the behaviour of SGNG because no more nodes are inserted and since the node-movements are not based on the local accumulated error. After 145000 iterations, 5.4.5(v) and 5.4.5(vi), it can be seen that SGNG and GNG have produced very similar node-placements, this is a result of the fact that no more nodes are being inserted and that both GNG and SGNG move nodes in the exact same manner. In



5.4.5(v) the nodes that were located about the jagged edge have been dragged away by the neighbouring nodes.

Figure 5.4.5 illustrates what happens if SGNG iterates for a long time. In the long run, the nodes spread out evenly because each node is affected by its neighbours. This implies that the supervised information that SGNG uses to place nodes, will be lost if SGNG iterates for too long, the exact number of iterations depends on the  $e_w$  and  $e_n$  values which control the node-movements. An interesting point is that the SGNG and GNG performed approximately the same on this distribution. Figure 4.5.6 shows the misclassification graphs.

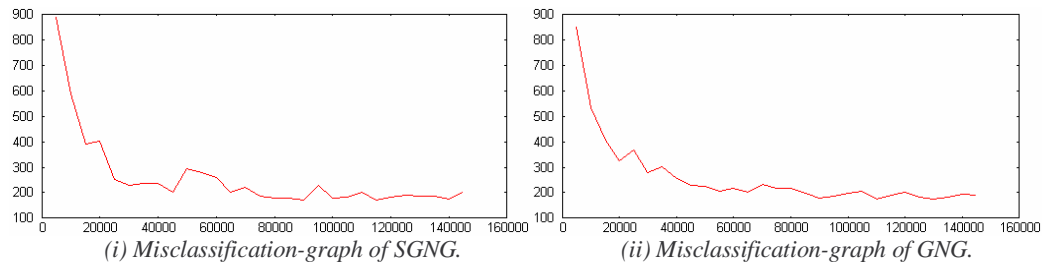


Figure 5.4.6. The misclassification-graphs of figure 5.4.5. Both graphs reach approximately the same low values. SGNG seems to reach lower values a little earlier than GNG.

As a complement, a second distribution (figure 5.4.7) with a fuzzier class-border will serve to further illustrate the differences in placements of nodes by the two algorithms GNG and SGNG and to illustrate the SGNG behaviour after the maximum number of nodes is reached.

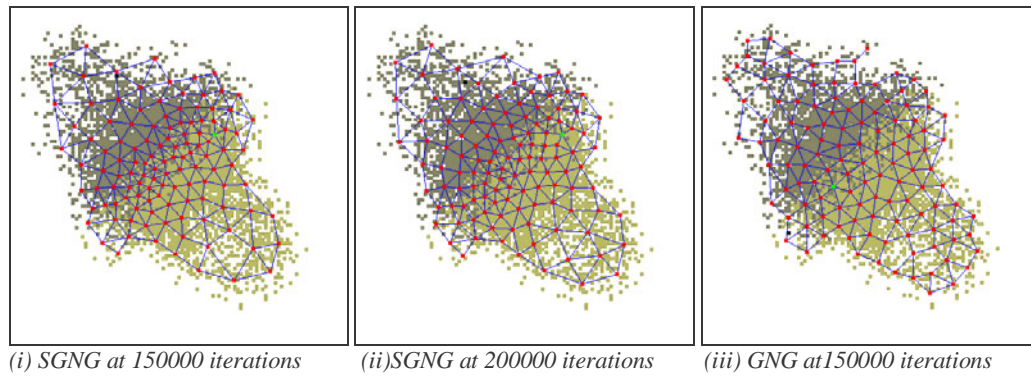


Figure 5.4.7. (i) SGNG at 150000 iterations with the maximum 120 nodes reached. (ii) SGNG at 200000 iterations, the nodes have “floated out”. (iii) GNG at 150000 with the maximum 120 nodes reached, GNG at 200000 iterations is extremely similar to GNG at 150000 iterations, and is therefore not shown. The parameters used were:  $max-nodes = 120$ ,  $\lambda = 800$ ,  $a_{max} = 70$ ,  $e_w = 0.02$ ,  $e_n = 0.0001$ ,  $\beta = 0.0125$ ,  $\eta = 0.03$ ,  $misclassification-back-log = 5000$ ,  $allowed\ number\ of\ misclassification = 0$ ,  $decision\ difference\ minimum = 0.0$ ,  $decision\ min\ limit = 0.65$ .

As can be seen in figure 5.4.7, the difference between SGNG and GNG at 150000 iterations is considerable, SGNG concentrates nodes where the decision problem is difficult, and GNG places nodes as to correspond to the input distribution. However, as the number of iterations increase, SGNG converges towards the node-placement of GNG and at 200000 iterations a hint of this can be detected. The convergence will take longer in this experiment since the  $e_w$  parameter is set lower than in the previous experiment. The performance of the two was again rather equal as can be seen in the misclassification-graphs in figure 5.4.8 below.

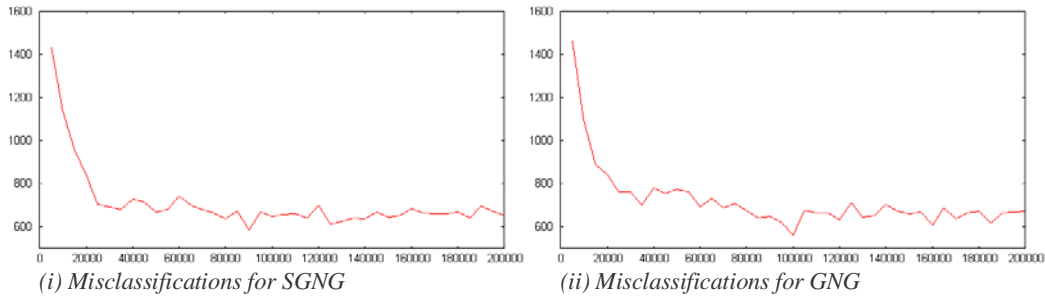
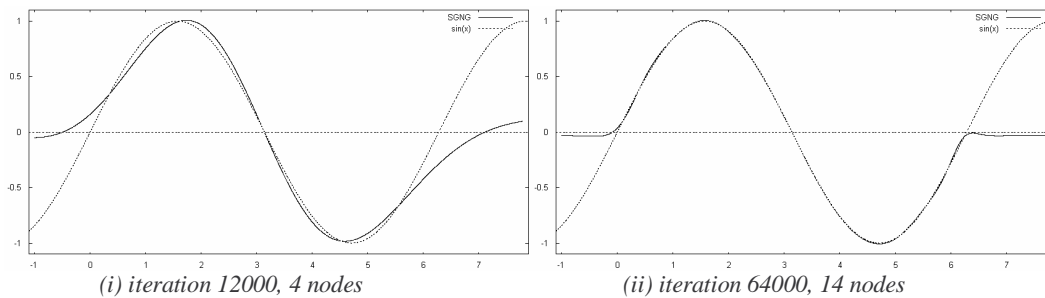


Figure 5.4.8. The misclassification graphs for the SGNG (i) and GNG (ii) experiments from figure 5.4.7. As can be seen their performance on the given task is close to equal. Again it can be seen that SGNG seems to reach lower values a little earlier than GNG, as can be observed in figure 5.4.6.

### The Sinus Function

In order to approximate the sinus function in the interval  $[0, 2\pi]$  a table of 100 uniformly distributed samples with corresponding inputs is used.  $\langle \text{input}, \text{output} \rangle$  pairs are then selected randomly with equal probability and presented to the SGNG-net. By experimenting with different values of the SGNG algorithm in connection with the sinus-approximation, the conclusion that rather a large value of the  $\lambda$  parameter is needed to get good placement of nodes. Figure 5.4.9 shows the sinus-approximation at three stages.



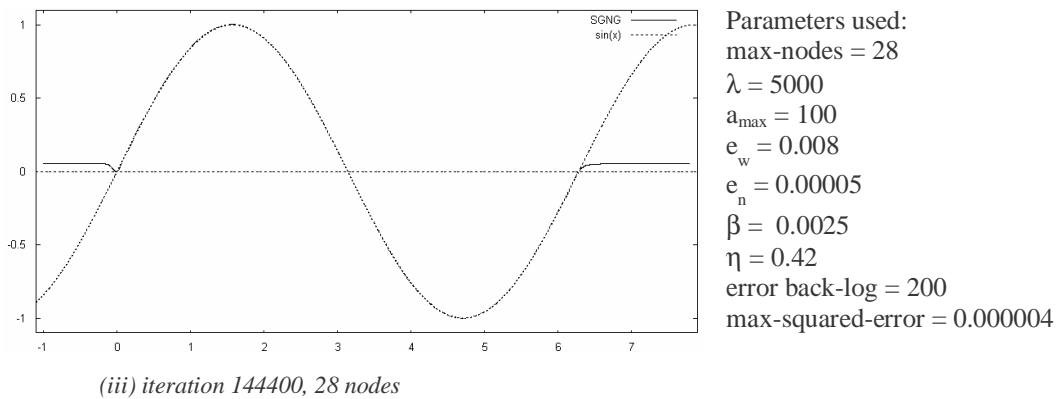


Figure 5.4.9 the sinus approximation at three stages. (i) 12000 iterations and 4 nodes. (ii) 64000 iterations and 14 nodes. (iii) 144400 iterations and 28 nodes.

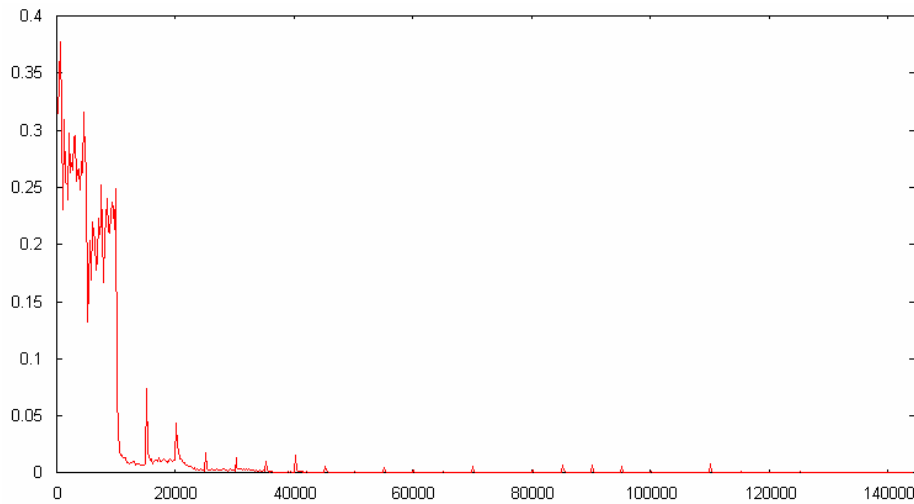


Figure 5.4.10. The mean squared output-error. After 144400 iterations using 28 nodes, the error is below the threshold  $4e-6$ . Looking closer at the error graph, spikes can be observed at even intervals, these spikes are due to the insertion of a new node, a result of the momentary disruption caused by the new untrained weights and the contribution of the new node's output.

## 5.5 Summary

SGNG can be successfully applied to both classification and function approximation tasks. The parameter settings in function approximation are a bit sensitive compared to the classification scenario. In function approximation, we require the output-values to be closer to the desired output than we do in classification since in classification we code the desired output as one-out-of-M vector and accept the answer where the greatest value in the actual output is larger than some threshold. Naturally, this would depend on the complexity of the classification task. However, generally this would be true.

A minor comparison was made between SGNG and GNG in the classification scenario. This comparison hints that the two approaches equal in performance. This comparison is not conclusive and suggested future work could be to more closely determine this, also, the comparison does not include the function

approximation scenario. In the task of classification, using SGNG instead of GNG, seem to be of little significance compared to the problem as a whole. GNG performs equally well in the experiments. SGNG reaches a lower misclassification-value earlier during execution but they both end up with approximately the same result.

A problem occurs with the SGNG algorithm when iterating for too long. The explanation is that since node-movements are based only on proximity to inputs and have no connection to the supervised error information, SGNG tend to converge to similar node placements as GNG, after sufficient time. This means that the supervised error information that was used to place the new nodes is meaningless in the end. This raises questions about the purpose of SGNG, however more experiments need to be conducted to decide upon this matter conclusively.

SGNG uses the supervised error information when updating the local error of only the winner-node. A future experiment might be to update the local errors using the derivative and to update not just the winner-node but also the rest of the nodes that contributed to the output. This would perhaps give a more complete and fair local-error update and in turn cause better placements of new nodes. However, this will not solve the long-term problem of SGNG converging towards GNG.

## 6 Implementation

### 6.1 Introduction

An implementation of the GNG, GNG-U and SGNG algorithms has been made in order to conduct experiments and investigate details of the various algorithms. C++ was the programming language chosen, mainly in order to allow fast object oriented code that would make testing new modifications easy and practical but also to allow incorporation of the code into other systems if the need arises. In general, the performance compared to the GNG implementation [6] is significantly better.

### 6.2 General Design Overview

A component-based approach was chosen in order to facilitate testing of new variants of parts of the algorithms. The GNG-U and SGNG implementations are built on the GNG implementation. GNG-U inherits GNG and overloads the appropriate methods with the addition of *utility*-specific methods. The SGNG implementation overloads only one method, the error update. All three algorithms are set up in the same way by means of plugging-in components that encapsulate desired behaviours. The next section will cover the different components by describing the set-up phase of any of the algorithm objects, simply “the algorithm object” since it is basically the same for all three. Figure 6.1 shows an overview of the design.

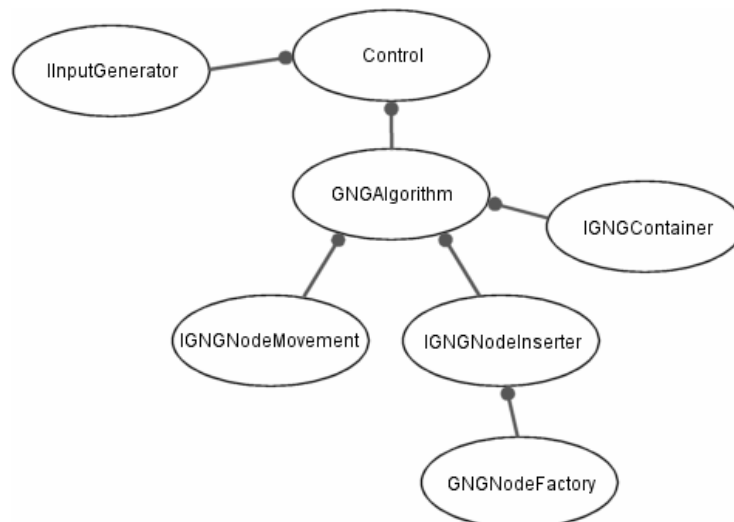


Figure 6.1. A simplified but accurate illustration of object interdependencies in the general case. In the case of SGNG the Control object (*RBFClassificationControl* or *RBFFunctionApproximationControl*) would also own an *RBFNetwork* object.

#### 6.2.1 The Components

As mentioned in the general design overview, a component-based approach was taken. Some of the behaviour has been encapsulated in different components, each

defined by an interface. The components are inserted into the Algorithm object, which uses the functionality provided by them.

The components of the algorithm objects are:

- *IGNGContainer* – describes the behaviour of a GNG container. The container creates and deletes nodes and edges. It is the main repository for edges and nodes. Creation of a node is done via a Factory-object to allow creation of user defined nodes. A default implementation, *DefaultGNGContainer*, exists. A reason for this component is for example SGNG where the RBF hidden layer is the *IGNGContainer*.
- *IGNGNodeInserter* – defines the manner in which nodes are inserted (by using the appropriate functionality of the current *IGNGContainer*) The GNG algorithm asks the current *IGNGNodeInserter* if it is ready to insert a node, if that is the case, a node is inserted in accordance with the current *IGNGNodeInserter*. A default fixed insertion implementation, *DefaultNodeInserter*, exists. There is also a *SGNGNodeInserter* used by the *SGNGAlgorithm* object.
- *IGNGNodeMovement* – defines how nodes are moved, the default implementation, *DefaultNodeMovement*, moves nodes in a linear fashion as described in the pseudo-code. Alternate movements, for example non-linear, could be interesting to implement but this is not included in this report.

### 6.2.2 Basic Elements

The basic elements consist of the *GNGNode* and *GNGEdge*. A *GNGNode* contains a reference vector, a utility, a local accumulated error and a set of edges to its neighbours. *GNGNodes* are created through the appropriate *GNGNodeFactory*. This allows for user defined *GNGNodes*, and is the mechanism used in the SGNG case where the *RBFNode*, derived from *GNGNode*, is created through the *RBFNodeFactory*. The factories are components of the *IGNGContainer* currently used. A *GNGEdge* is simply an association of two *GNGNodes* but also contains the age for that particular edge.

### 6.2.3 Input Generators

Input generators are derived from *IInputGenerator* and can be of two types, *FunctionGenerator* or *ClassesGenerator*. Regardless of the type of the input generator it will work properly in the case of GNG and GNG-U, the type only matters in the case of SGNG since it is combined with a RBF network that needs outputs when trained. The input generator type decides which control object is used in the SGNG case. If the input generator is a *FunctionGenerator* then *RBFFunctionApproximationControl* will be used and if it is a *ClassesGenerator* then an *RBFClassificationControl* will be used. Control objects will be described below.

## 6.2.4 Algorithm Classes

There are three algorithm objects; the *GNGAlgorithm*, *GNGUAlgorithm* and the *SGNGAlgorithm*. The *GNGUAlgorithm* and *SGNGAlgorithm* are both derived from *GNGAlgorithm* and override the appropriate methods in order to define their respective behaviour.

## 6.2.5 Control Objects

The control objects perform the over-all control, such as presenting input signals and iterating the algorithm objects, and in the *SGNG* case, handles the RBF network input presentation and training. They also define the stop criteria, and in the *SGNG* case of classification and function approximation, there is a minor difference in the two. In classification, it is normally more interesting to know when the number of misclassifications reaches a certain allowed maximum and in function approximation, the criteria are based on the output squared-error.

There are three control objects:

- *GNGControl* – Controls the *GNG* and *GNG-U* algorithm iteration, by presenting an input and iterating the algorithm.
- *RBFClassificationControl* – Controls the *SGNG* (with RBF) in a pattern classification scenario as previously discussed in section 5.2.1. The outputs are coded in a one-out-of-M manner.
- *RBFFunctionApproximationControl* – Controls the *SGNG* (with RBF) in a function approximation scenario.

## 6.2.6 Graphical Representation

An Open-GL graphical representation is provided for two-dimensional generators, this is helpful when attempting to understand the general behaviour of the three algorithms. However, since this report is not mainly about the implementation, details concerning this are not included in the report.

## 6.3 User Instructions

The program uses an initialisation-file named *gng.ini*. The initialisation-file contains all the parameters for all three algorithms. All parameters must be present and in the order described below. Comments are started by '#' and forces the program to ignore the rest of that line. This is an example of a *gng.ini* file.

```
# MODE (gng, gngu, sgng)
gng

# Init from distribution (0=no or 1=yes)
0

#####
### Representation
#####
```

```

# REPRESENTATION_UPDATE
#   updates the graphics every n iterations
10
# ITERATION_DELAY_IN_MILLISECONDS
#   delay in milliseconds between each iteration
0

#####
### GNG settings
#####

# GNG MSE BackLog
#   GNG MSE is logged to file during execution.
#   the mean is calculated every n iterations.
500
# MAX NODES
60
# INSERT EVERY (lambda)
600
# NEW NODE-POSITION
#   default is 0.5, this is not an original parameter in GNG.
#   This is only for test purposes.
0.5
# ERROR DECAY for INSERTION (alpha)
#   IMPORTANT: in sgng-mode this should be 0.5
#               according to SGNG-specs
0.5
# AGE MAX, maximum allowed age of an edge before it is removed.
80
# MOVE WINNER (epsilon winner)
0.05
# MOVE NEIGHBOR (epsilon neighbor)
0.0006
# ERROR DECAY (beta)
0.0005

#####
### GNG-U specific
#####

# UTILITY DECAY
#   this is not an original parameter in GNG, utility decay
#   should equal error decay. This is only for test purposes.
0.0005
# UTILITY BOUND (k)
1.0

#####
### SGNG (GNG with RBF) specific
#####

# MISCLASSIFICATION or ERROR BACKLOG
#   misclassification or error is logged to file in steps of n
100
# NUM MISCLASSIFICATIONS ALLOWED in backLog steps
#   if the input distribution is a classification task then this
#   parameter determines the maximum number of allowed
#   misclassifications during the backlog, before stopping.
0
# CLASSIFICATION decision diff minimum.
#   the minimum difference between the best and second
#   best answers in the one-out-of-k response vector.

```



```

# a way of forcing more sure answers (if possible).
# (a value between 0..1)
0.0
# CLASSIFICATION decision min limit.
# The best answer must be greater or equal to this
# limit to be accepted (a value between 0..1)
0.7
# MAX SQUARED ERROR
# if the input distribution is a function approximation task
# then this parameter determines the maximum allowed error before
# stopping.
0.00001
# ETA, step-size used in weight training
0.2

#####
### Generator stuff.
#####

# DIM, the Default generator uses this setting.
2
# DISTRIBUTION_TYPE
1

#####
### Distribution types
#####
# 0 - Default
# 1 - Four of default
# 2 - Jump Default
# 3 - Jump Square moving default
# 4 - Square Slow Moving Default
# 5 - Square Moving Four of Default
# 6 - Four of Square Moving Default
# 7 - Four of Four of Default
# 8 - UpDown Moving of Default
# 9 - UpDown Moving, Square Moving of Default
# 10 - Square Moving, UpDown Moving, of Default
# 11 - TwoClasses - classification
# 12 - XOR function - function approximation
# 13 - Discrete - classification
# 14 - Sinus function - function approximation

```

To be able to run the program, GLUT32.dll (in unix, the appropriate libs) must exist since the program relies on GLUT/OpenGL for representation.

Run-time usage is quite simple, there are only three possible interaction possibilities: left-mouse button starts and pauses execution. When in paused mode the right and middle mouse buttons allow single step and REPRESENTATION\_UPDATE steps update of the representation.

To end the program, simply close the window.

## References

- [1] MacQueen, J. *Some methods for classification and analysis of multivariate observations*. In LeCam, L. and Neyman, J., editors, Proceedings of the Fifth Berkeley Symposium on Mathematical statistics and probability, volume 1, Berkeley. University of California Press, 1967, pp. 281-297.
- [2] Kohonen, T. *Self-organized formation of topologically correct feature maps*. Biological Cybernetics, 1982, pp. 43:59 - 69.
- [3] Martinetz, T. M. and Schulten, K. J. A “neural-gas” network learns topologies. In Kohonen, T., Mäkisara, K., Simula, O., and Kangas, J., editors, *Artificial Neural Networks*, North-Holland, Amsterdam, 1991, pp. 397-402.
- [4] Fritzke, B. *A growing neural gas network learns topologies*. In G. Tesauro, D. S. Touretzky, and T. K. Leen, editors, *Advances in Neural Information Processing Systems 7*, MIT Press, Cambridge MA, 1995, pp. 625-632.
- [5] Martinetz, T. M. *Competitive Hebbian learning rule forms perfectly topology preserving maps*, ICANN’93: International Conference on Artificial Neural Networks, Springer, Amsterdam, 1993, pp. 427-434.
- [6] Loos, H. S. and Fritzke, B. *DemoGNG v1.5*, 1998
- [7] Fritzke, B. *Fast learning with incremental RBF networks*. Neural Processing Letters, 1(1):2-5, 1994.
- [8] Fritzke, B. *A self-organizing network that can follow non-stationary distributions*, Proc. of the International Conference on Artificial Neural Networks '97, Springer, 1997, pp. 613-618.
- [9] Haykin, S., *Neural Networks A Comprehensive Foundation* 2<sup>nd</sup> ed., Prentice Hall Inc. 1999, p. 2.
- [10] Hassoun, M. H. *Fundamentals of Artificial Neural Networks*, MIT Press, Cambridge, MA, 1995, pp. 199:289.