Growing Neural Gas Efficiently

Daniel Fišer, Jan Faigl, Miroslav Kulich

Department of Cybernetics, Faculty of Electrical Engineering, Czech Technical University in Prague, Technická 2, Prague 6, Czech Republic 166 27

Abstract

This paper presents optimization techniques that substantially speed up the Growing Neural Gas (GNG) algorithm. The GNG is an example of the Self-Organizing Map algorithm that is a subject of an intensive research interest in recent years as it is used in various practical applications. However, a poor time performance on large scale problems requiring neural networks with a high amount of nodes can be a limiting factor for further applications (e.g., cluster analysis, classification, 3-D reconstruction) or a wider usage. We propose two optimization techniques that are aimed exclusively on an efficient implementation of the GNG algorithm internal structure rather than on a modification of the original algorithm. The proposed optimizations preserve all properties of the GNG algorithm and enable to use it on large scale problems with reduced computational requirements in several orders of magnitude.

Keywords: Self-Organizing Map, Growing Neural Gas, Nearest Neighbor Search, 3-D Surface Reconstruction

1. Introduction

The Growing Neural Gas (GNG) [1] is one of Self-Organizing Map (SOM) [2] algorithms for unsupervised learning. Unsupervised learning (or sometimes called learning without teacher) is a learning method that works solely with an input data and has no information about the desired output. Input data are consecutively presented to SOM in a form of input signals and SOM changes its topological structure to fit to the input data using its own specific mechanism – self adaptation. The GNG and other growing neural networks (such as Growing Cell Structures [3], Growing Grid [4], etc.) use furthermore a growing mechanism for a gradual adaptation and a self adjusting of its own size. The growing neural network starts in some minimal state (e.g., with some minimal number of neurons in the network), which is adapted to input data. Then, it grows (increases its size) and adapts again. This cycle is repeated until a desired resolution of the neural network is reached.

SOM algorithms are used in various applications such as vector quantization [5, 6, 7], cluster analysis [1, 3, 8, 9, 10], classification [11, 6], and 3-D reconstruction [12, 13, 14, 15], but a poor time performance, especially on large scale problems, can be a limiting factor for further applications or a wider usage. Although several approaches addressing the computational requirements have been proposed [16, 17, 7], we have found out that the GNG algorithm can be implemented in a more efficient way than a direct implementation of the original description. Moreover, based on our application of the GNG algorithm in a real problem of 3D reconstruction [18][1], we identified the most time consuming operations of the GNG algorithm and proposed optimization techniques significantly reducing the real required computational time. Hence, the goal of this paper is to show how to overcome the issue of the poor time performance and how to implement the GNG algorithm with optimizations providing a significant speedup.

The paper is organized as follows. First, a detailed description of the GNG algorithm is presented in the next section to identify and understand its most time-consuming parts. In Section 3, an overview of the related work is presented. Sections 4 and 5 are dedicated to description of the speedup techniques proposed. The real benefit of the techniques is evaluated in Section 6, the discussion of the experimental results is presented in Section 7, and the concluding remarks are presented in Section 8.

2. The Growing Neural Gas

A GNG network structure is a graph consisting of a set of nodes and a set of edges connecting the nodes. Each node has associated a weight vector corresponding to the node's position in the input space and an error variable intended for identification of the parts of the network least adapted to input signals. Each edge is unambiguously

Email addresses: danfis@labe.felk.cvut.cz (Daniel Fišer), xfaigl@labe.felk.cvut.cz (Jan Faigl), kulich@labe.felk.cvut.cz (Miroslav Kulich)

[1] Videos with visualization of the 3-D reconstructions are available at http://www.youtube.com/watch?v=yoPcZpCPfyI, http://www.youtube.com/watch?v=oXx3oJ8omOQ, http://www.youtube.com/watch?v=j_t8LkAXS9Q.

Table 1: Notation

G	the set of all nodes in the network
ν, μ	nodes
N_ν	the set of all topological neighbors of node ν
$\vec{w}_\nu$	the weight vector of a node ν
E_ν	the error of a node ν
$A_{\nu,\mu}$	the age of the edge between the nodes ν and μ
c	the cycle counter
s	the step counter

identified by a pair of nodes. The schema of the GNG is depicted in Algorithm 1 with supporting functions in Algorithm 2 and notation used through this paper can be seen in Table 1.

The GNG works as follows. After initialization, which places two randomly generated nodes into a network, two main phases are alternating until a selected stopping criterion is met. The first phase ("self-organizing") is adaptation, which is performed in λ steps. In each step, random input signal is generated and the neural network adapts itself to it: a connection between two nodes nearest to the input signal is strengthened (or created if it does not exist), then the nearest node and all its topological neighbors (nodes connected directly to the node by an edge) move towards the input signal and the nearest node's error is increased. This helps to identify areas where nodes are not sufficiently adapted to input signals. After that, the aging mechanism of edges is triggered – those edges that were not strengthened for a long time (the age of the edge is higher than A_{max}) are removed from the network. In the last step of the adaptation, an error of each node is decreased. Using this mechanism the neural network "forgets" old errors and thus it can focus on the most recent ones.

In the second phase ("growing"), a new node is created and connected into the network. The node's error is used for an identification of the area where the adaptation was least successful – the node with the largest error and its neighbor with the largest error are found. A new node is created at the halfway between them. The errors of those nodes are decreased.

2.1. An alternative formulation of the GNG algorithm

We had observed that the original description of the GNG made by Bernd Fritzke in [1] can be reformulated without changes of the algorithm behavior, i.e., the new algorithm works exactly in the same way as the original one. The reformulation can be considered as "cosmetic"; however, it allows a more straightforward application of the proposed optimizations of the time consuming operations. Thus, the alternative formulation helps in further explanation of the speedup techniques proposed, and therefore, this section is dedicated for description of differences between the original and the alternative algorithm implementation.

Algorithm 1: The original Growing Neural Gas algorithm

```
GNG()
1  initialize the set G by two nodes with random weight vectors
2  c ⟵ 0
3  s ⟵ 0
4  ξ⃗ ⟵ random input signal
5  s ⟵ s + 1
6  ν, μ ⟵ TWO_NEAREST_NODES(ξ⃗)
7  foreach n in N_ν
8      A_{n,ν} ⟵ A_{n,ν} + 1
9  INC_ERROR(c, s, ν, ‖w⃗_ν − ξ⃗‖²)
10 w⃗_ν ⟵ w⃗_ν + ε_b(ξ⃗ − w⃗_ν)
11 w⃗_n ⟵ w⃗_n + ε_n(ξ⃗ − w⃗_n), ∀n ∈ N_ν
12 create an edge between ν and μ if it does not exist
13 A_{ν,μ} ⟵ 0
14 foreach a, b in all edges in map
15     if A_{n,ν} > A_max
16         delete edge connecting n and ν and all nodes w/o edges
17 if s = λ
18     GNG_NEW_NODE(c)
19     c ⟵ c + 1
20     s ⟵ 0
21 DEC_ALL_ERROR(β)
22 if stopping criterion is met
23     terminate algorithm
24 else
25     go to step 4.
```

Algorithm 2: Functions for the original GNG

```
INC_ERROR(c, s, ν, v)
1  E_ν ⟵ E_ν + v

DEC_ERROR(c, ν, α)
1  E_ν ⟵ αE_ν

SET_ERROR(c, ν, v)
1  E_ν ⟵ v

DEC_ALL_ERROR(β)
1  E_n ⟵ βE_n, ∀n ∈ G

LARGEST_ERROR(c)
1  q ⟵ arg max_{n∈G} E_n
2  f ⟵ arg max_{n∈N_q} E_n
3  return q, f
```

The original formulation of the GNG algorithm is depicted in Algorithm 1 and our formulation in Algorithm 3. The alternative formulation of the algorithm consists of three parts that logically belong together. The creation of a new node is separated into dedicated function GNG_NEW_NODE(). The function GNG_ADAPT() performs one iteration of adaptation to a single input signal. Finally, the main function GNG() wraps the whole algorithm into one cycle within which a stopping criterion is checked. In the original formulation, the whole algorithm run in one cycle that adapts a network to an input signal in each step and every λ steps a new node is inserted. This is equivalent to λ adaptation steps followed by a node insertion as it is in our formulation.

The aging of edges was originally defined in two cycles. The first one that increases ages of edges incidenting with winning node and the second one that removes all edges

Algorithm 3: The Growing Neural Gas algorithm

```
GNG()
1  initialize the set G by two nodes with random weight vectors
2  c ⟵ 0
3  while stopping criterion is not met
4      for s ⟵ 0 to λ − 1
5          ξ⃗ ⟵ random input signal
6          GNG_ADAPT(c, s, ξ⃗)
7      GNG_NEW_NODE(c)
8      c ⟵ c + 1

GNG_ADAPT(c, s, ξ⃗)
1  ν, μ ⟵ TWO_NEAREST_NODES(ξ⃗)
2  INC_ERROR(c, s, ν, ‖w⃗_ν − ξ⃗‖²)
3  w⃗_ν ⟵ w⃗_ν + ε_b(ξ⃗ − w⃗_ν)
4  w⃗_n ⟵ w⃗_n + ε_n(ξ⃗ − w⃗_n), ∀n ∈ N_ν
5  create an edge between ν and μ if it does not exist
6  A_{ν,μ} ⟵ 0
7  foreach n in N_ν
8      A_{n,ν} ⟵ A_{n,ν} + 1
9      if A_{n,ν} > A_max
10         delete edge connecting n and ν and all nodes w/o edges
11 DEC_ALL_ERROR(β)

GNG_NEW_NODE(c)
1  q, f ⟵ LARGEST_ERROR(c)
2  w⃗_r ⟵ (w⃗_q + w⃗_f)/2
3  delete an edge connecting q and f and create two new edges
   between r and q and between r and f
4  DEC_ERROR(c, q, α)
5  DEC_ERROR(c, f, α)
6  SET_ERROR(c, r, (E_q + E_f)/2)
```

with age higher than threshold and thus requires iteration over all edges in the graph. We have reformulated this mechanism so that edges are removed in the same cycle an increase of the edge's age takes place (see lines 6 to 10 in GNG_ADAPT() function). It can be done so because the age is not changed anywhere else so edges can obsolete only in that cycle. The difference between these two formulations is that the age of the activated edge is effectively set to value 1 instead of 0.

More noticeable difference between the formulations is that we have moved the decreasing of all error variables from the end of the main algorithm cycle to the end of the adaptation step (compare line 21 in Algorithm 1 and line 11 in GNG_ADAPT() function in Algorithm 3). The difference is that in the original formulation the decreasing is performed after an insertion of a new node whereas in our formulation, the decreasing is performed before an insertion. This difference can be compensated by a selection of the parameter α so that $\alpha = \beta\alpha_{orig}$.

The proposed alternative formulation has been considered in a real problem of 3D reconstruction [18] in which we identified two most time consuming operations of the algorithm. The first operation is the nearest neighbor search performed in each adaptation step. The second operation is the handling of node errors, i.e., the decreasing of all errors at the end of each adaptation step and the search for the node with the largest error when a new node is created. All other operations, except these, can be performed in a constant time[2].

The problem of search operation has been addressed in several GNG based approaches that are described in the next section. Our approach for this issue is then presented in Section 4. A speedup technique to address the error handling is proposed in Section 5.

3. Related Work

Several contributions were made in the area of speeding up the GNG algorithm in recent years. In these contributions, two main approaches can be identified. The first approach aims to decrease the computational burden of the GNG using supporting structures and it can be considered as an incremental extensions without affecting the original underlying GNG mechanism. The second approach requires modification of the GNG, and therefore, it is less general as it modifies the internal mechanism of the GNG. The representative and the most promising approaches are briefly described in this section.

The Incremental Growing Neural Gas (IGNG) [16] algorithm changes the mechanism of the creation of a new node by introducing an "embryo" node and a "mature" node. An "embryo" node is created whenever the presented input signal is farther from the established network than a predefined threshold. The "embryo" node later becomes the "mature" node after a predefined number of excitations via input signals. An output network is formed exclusively by the "mature" nodes. This approach deals with the second aforementioned most time-consuming part of the GNG by replacing it with a different mechanism of the node creation. The IGNG does not need to compute errors of the nodes because the node creation is based solely on the distance of the input signal from the established network. The nearest neighbor search operation is not targeted in this work at all.

The Density Based Growing Neural Gas (DB-GNG) [17] uses similar approach for a node creation as the IGNG. If an input signal is farther from the established network than a predefined threshold, a new node is created, but only if the region, where the new node would be inserted, has a sufficient density (i.e., the region was sufficiently sampled by input signals so far). The DB-GNG uses a combination of slim-tree [19], dbm-tree [20], and r-tree [21] as a supporting structure for the nearest neighbor and range search, which helps to speed up the algorithm. The handling of node errors is addressed by a new mechanism of a node creation and, in contrast to the IGNG, the nearest neighbor search is challenged by the application of an enhanced search structure.

[2]Beside the complexity analysis, a real profile of the algorithm run shows that the nearest neighbor search constitutes about 48% of the algorithm runtime and the search for the node with the largest error approximately 51%; hence, almost all computational time is spent in these two operations.

The Growing Neural Gas with targeting (GNG-T) [7] avoids an expensive error decreasing at the end of the adaptation cycle by introducing a new parameter T that represents a target average error over all nodes. At the start of the adaptation phase, all errors are set to zero. Errors of nodes constantly grow over all adaptation steps, i.e., errors of all nodes are accumulated over the whole adaptation phase without any decay. After the adaptation phase, the network grows (a new node is added) if the average error over all nodes is higher than the selected value T, or the network shrinks (the node with the highest error is removed) if the average error is below T. The nearest neighbor search is not addressed in [7].

All aforementioned approaches address the problem by introducing a brand new algorithm to avoid a dealing with problematic parts of the GNG. The approach presented in this paper, on the contrary, aims directly on the original GNG algorithm, and therefore, it preserve all the properties, including generality, just using a more sophisticated implementations of the needed operations. Hence, this paper can be considered as a tutorial how to implement the GNG algorithm more efficiently.

4. Nearest Neighbor Search

A naïve and straightforward implementation of the nearest neighbor search is a linear search, which is also the slowest one because it requires iteration over all nodes each time a search is performed. Therefore, the searching should be based on a more sophisticated algorithm using some supporting structure.

K-d tree [22] is a well-known structure for the nearest neighbor search based on partitioning of a space into a binary tree using splitting axis-aligned hyperplanes. The time complexity of the search query for two nearest neighbors can be reduced to $O(n^{1-1/k})$ in the worst case, where k is the dimensionality of the space and n is the number of nodes in the tree. A disadvantage of this structure is the cost of the node update that has, even in a balanced k-d tree, complexity $O(\log n)$. Since nodes are constantly moving in the GNG, the update operation must be performed several times each step (not to mention re-balancing of the tree).

The vantage point tree (vp-tree) [23, 24] is another search structure, which partitions a space on the basis of a distance from a chosen point (vantage point). This algorithm can reach $O(\log n)$ expected time for a search query under certain circumstances but a disadvantage is, again, update operation, which is relatively expensive. Other search algorithms such as the r-tree [21], slim-trees [19], or dbm-tree [20] also suffer of similar disadvantages.

Regarding the issue of the standard structures, we propose a different approach, called *Uniform Grid*. It ensures a constant time for the update operation and, as indicated by experimental results, a near constant time for the search query.

4.1. Uniform Grid

The idea of the considered *Uniform Grid* (UG) searching algorithm is based on speeding up the searching process by dividing and indexing the search space. More precisely, only the Euclidean search space (L_2-norm) is considered. So, having a point its coordinates are used as indexes to a part of the search space where a local search for the nearest neighbors is performed. Although in the worst case the whole search space is examined for a single query, real computational requirements are significantly lower. The proposed techniques is similar to the "*bucketing*" method for the point-location problem [25], which has average complexity $O(1)$, even though the worst case complexity is $O(n)$ for n points.

The idea of the uniform grid is relatively straightforward; however, the key issues are how to select the grid dimensions, its resolution, and how to perform the local search effectively to reduce the real required computational time. On the other hand, the method is general enough to be used not only for 2-D or 3-D cases, but also for a higher dimensions of the searching space. Therefore, herein presented formal description of UG is for a D dimensional space, but without lost of generality the illustrations of the algorithm's structures are shown only for the 2-D case for a better readability. First, we assume the parameters of the grid are known in advance to simplify the description and correctness of the searching procedure. Later, a self adjusting procedure is introduced in Section 4.2.

Let $\vec{g} = (g_1, g_2, ..., g_D)$ be the proportions of the grid, i.e., the total number of the grid cells is $n = g_1 g_2 ... g_D$, l be the length of a single cell side, and $\vec{o} = (o_1, o_2, ..., o_D)$ be the origin of the covered input signal space. Each cell C of the uniform grid $UG = g_1 \times g_2 \times \dots g_D$ has associated list of nodes whose weight vectors are encapsulated by the cell. Then, the coordinates $\vec{p}_\nu = (p_1, p_2, ..., p_D)$ of the cell C within the grid where a node ν is located can be computed using the node's weight $\vec{w}_\nu = (w_1, w_2, ..., w_D)$ as

$$\vec{p}_\nu = \left\lfloor \frac{\vec{w}_\nu - \vec{o}}{l} \right\rfloor, \tag{1}$$

where $\lfloor \cdot \rfloor$ denote the floor function.

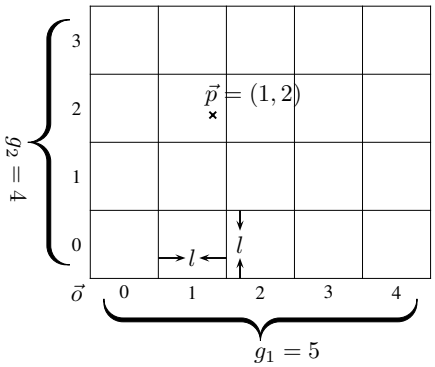
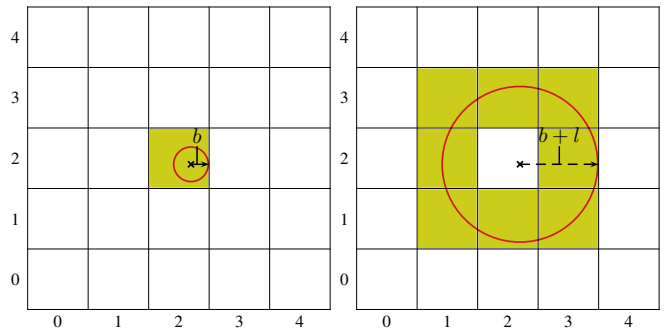
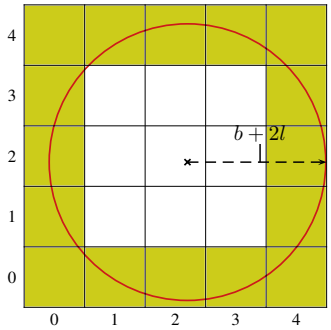


Figure 1: An example of 2-D uniform grid.



(a) step 1, boundary distance b, initial cell C is searched

(b) step 2, $b + l$, cells within the radius 1 from C are searched



(c) step 3, $b + 2l$, cells within the radius 2 from C are searched

Figure 2: An example of searching in 2-D grid: (a) In the first step only one cell is searched, (b) then all cells around it (in radius 1); (c) then all cells in radius 2, etc. The cross represents the input signal, red circle the boundary distance b, and yellow cells are the cells that are currently searched.

It is obvious from Eq. (1) that for a given D the position of ν in the grid can be computed in a constant time; hence, a node can be inserted into the grid in a constant time as well as the node update, which can be made as consecutive remove and insert operations. An example of 2-D uniform grid is shown in Fig. 1.

A procedure for finding the closest nodes is a bit more complicated as it has to find up two nearest neighbors. First, a corresponding cell C is determined for an input signal $\vec{\xi} = (\xi_1, \xi_2, ..., \xi_D)$ using Eq. (1). The closest node can be in the same cell C; however, due to the rounding of the node indexes, it may also be in the next cell (or other cells because its distance from the node can be longer than l). Therefore the minimal orthogonal distance b to the C's border is computed according to Eq. (2).

$$b = \min_{i=1...D} \min\left(|\xi_i - o_i - p_i l|, |\xi_i - o_i - (p_i + 1)l|\right) \quad (2)$$

Then, an iterative procedure is performed to find two nearest nodes. The procedure starts with the cell C and uses the distance b to ensure that there are not closer nodes in next cells. If C contains two nodes that are in less distance than b, then other nodes can be only within longer distance. If such nodes are not found, the distance b is increased by the length of the cell side l:

$$b' = b + l \quad (3)$$

and cells within the radius 1 from C are searched. This procedure is repeated until the nearest nodes are found, or the whole UG is searched. Each time of the repeat, the cells within a longer radius (increased about 1) are considered, and therefore, each cell of the grid is examined once (at maximum), see example of searching sequence depicted in Fig. 2.

Discussion – The procedure finds two nearest nodes (if exist) and in the worst case the complexity of the search query is $O(n)$. Thus, the proposed procedure is not worse than a linear search. However, the real required computational time of the search query depends on the occupancy of the cells in the grid. If the cells are sparsely occupied, i.e., many cells are empty and the rest of the cells contain a small amount of nodes, then a large number of cells must be searched and the search query is slow because a large part of the grid has to be examined. On the other hand, if the cells are occupied too much densely, i.e., each cell contains a large amount of nodes, only a small part of the grid is searched but a large amount of nodes must be checked within a single cell, which also leads to the linear search algorithm. An appropriate balance between these two extremes is required for the best performance of the uniform grid, but unfortunately it depends on the grid parameters such as the proportions ($\vec{g}$) and the cell side length (l), which must be known in advance. It can also be hard to find the ideal parameters because the neural network gradually grows; hence, the occupancy of the grid is changing in time. Therefore, we addressed these issues by a dynamic version of the uniform grid, which is described in the next section.

4.2. Growing Uniform Grid

A proper selection of the uniform grid parameters can be avoided using a mechanism of self adjusting to a node distribution. The idea of the proposed *growing uniform grid* is based on observations of the performance of the above described uniform grid. The uniform grid is a static structure that does not scale well with the growing number of nodes in the network. If the proportions of the grid is chosen to work fast on a small amount of nodes then the search query will be slow when the number of nodes in the network cross certain threshold. On the other hand, if the proportions are chosen to better fit a large amount of nodes, the algorithm will be slow at the beginning when the network is sparse (and possibly will slow again when the network grows over certain limit). Therefore, the dynamic version of the algorithm needs a mechanism to adjust itself to the current size of the network.

Let h_d denote a density of the uniform grid, i.e., an average number of nodes per cell. Let h_t denote an allowed maximal density and let h_ρ denote an expansion factor, i.e., the ratio of the number of nodes in a new network

and the number of nodes in the old network. The growing uniform grid starts with a single cell encapsulating an axis aligned bounding box of the input signals. Once $h_d > h_t$ a brand new uniform grid is built consisting of h_ρ-times more cells than the current grid (proportions and the cell side length are computed appropriately). After that, all nodes are inserted into the new grid. An example of the growing grid is depicted in Fig. 3. The rebuilt of the uniform grid is made in a linear time because the insertion of a single node can be performed in a constant time. Thus, the complexity of the growing uniform grid is still $O(n)$ as of the original uniform grid, but the growing uniform grid can scale better to the growing network if appropriate values of parameters are chosen.

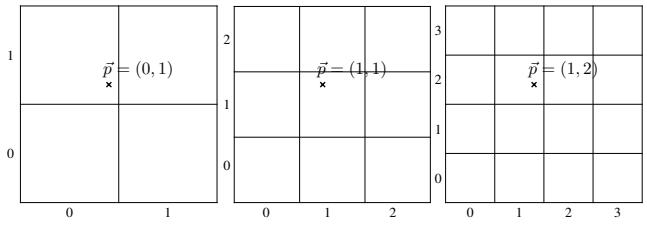


Figure 3: A growing of the uniform grid (from left to right) covering the same space and containing the same node.

The selection of the appropriate values of h_t and h_ρ can be still a somehow difficult task because the values, obviously, depend on the distribution of the input signals. Based on experimental results, we estimated the values of h_t and h_ρ to 0.1 and 1.5, respectively. These values are used in all experiments described in Section 6.

Note that the decision on rebuilding of the grid can be based on a different measure than the average number of nodes per cell. It can be the median, mode, or any other measure. We have chosen the average measure because it is fast and easy to compute and it was able to provide sufficient performance in the experimental evaluation.

5. Error Handling

The second most time consuming part of the GNG algorithm is the handling of node errors, above all, the decreasing of all errors at the end of the adaptation step (DEC_ALL_ERROR function) and the finding of a node with the largest error at the beginning of the "growing" phase (LARGEST_ERROR function). A naïve implementation of both operations examine all nodes in the network, and therefore, its complexity is linear. Nevertheless, we propose a mechanism for each operation that significantly reduces its required computational time. The mechanism is based on observations how the error values evolve during the adaptation and growing phases.

5.1. Adaptation Phase

The GNG algorithm is performed in cycles where the adaptation and the growing phases are alternating. The cycles are counted by the counter denoted by c. The adaptation phase is performed in λ steps, which are counted by the counter denoted by s counting from 0 to $\lambda - 1$ (see Algorithm 3).

The DEC_ALL_ERROR function with a linear complexity represents the most expensive error handling of the adaptation phase. Therefore, we focused on a more sophisticated implementation to reduce the computational burden. To do so, we remove the function from the algorithm and examine the algorithm in order to propose a mechanism that will replace it.

During the adaptation phase, a node error is changed only in the INC_ERROR function, which basically adds only a given value to the error variable. Let c_j denote the current cycle of the algorithm. Let E_{ν,c_0} denote the value of the error of the node ν at the end of the cycle c_0 ($c_0 \leq c_j$). Following cases for the error can happen.

If the error is not increased whatsoever in the current cycle (c_j), the error at the end of the cycle has value

$$E_{\nu,c_j} = \beta^{(c_j - c_0)\lambda} E_{\nu,c_0} \tag{4}$$

because the error is decreased λ-times in each adaptation phase between c_0 and c_j (including c_j and excluding c_0).

If the error is increased exactly once by the value v_1 in the step s_1 (of cycle c_j), the value of the error can be computed step by step as follows.

1) According to Eq. (4), the error at the end of the *previous* cycle has value

$$E_{\nu,c_j-1} = \beta^{[(c_j-1)-c_0]\lambda} E_{\nu,c_0}. \tag{5}$$

2) The error is decreased s_1-times to have the following value in the step s_1:

$$E'_{\nu,c_j} = \beta^{s_1} E_{\nu,c_j-1}. \tag{6}$$

3) The error is increased by v_1:

$$E''_{\nu,c_j} = E'_{\nu,c_j} + v_1. \tag{7}$$

4) Finally, the error is decreased $(\lambda - s_1)$-times to get the actual value at the end of the cycle c_j:

$$\begin{aligned} E_{\nu,c_j} &= \beta^{\lambda - s_1} E''_{\nu,c_j} \\ &= \beta^{\lambda - s_1}(\beta^{s_1}\beta^{[(c_j-1)-c_0]\lambda} E_{\nu,c_0} + v_1) \\ &= \beta^{(c_j-c_0)\lambda} E_{\nu,c_0} + \beta^{\lambda - s_1} v_1. \end{aligned} \tag{8}$$

Similarly, if the error is increased two times by v_1, v_2 in steps s_1, s_2 ($s_1 < s_2$), respectively, its value is

$$\begin{aligned} E_{\nu,c_j} &= \beta^{\lambda - s_2}[\beta^{s_2 - s_1}(\beta^{s_1}\beta^{[(c_j-1)-c_0]\lambda} E_{\nu,c_0} + v_1) + v_2] \\ &= \beta^{(c_j-c_0)\lambda} E_{\nu,c_0} + \beta^{\lambda - s_1} v_1 + \beta^{\lambda - s_2} v_2. \end{aligned} \tag{9}$$

This can be generalized for n changes v_1, v_2, ..., v_n in steps s_1, s_2, ..., s_n ($s_1 < s_2 < ... < s_n$):

$$E_{\nu,c_j} = \beta^{(c_j-c_0)\lambda} E_{\nu,c_0} + \sum_{i=1}^{n} \beta^{\lambda - s_i} v_i. \tag{10}$$

Algorithm 4: Functions of the proposed improvements of the GNG. Note that all powers of β can be pre-computed into an array and the expensive processor operation can be avoided.

```
INC_ERROR*(c, s, ν, v)
1  FIX_ERROR(c, ν)
2  E_ν ⟵ β^{λ−s} E_ν + v
3  update the node ν in the heap

DEC_ERROR*(c, ν, α)
1  FIX_ERROR(c, ν)
2  E_ν ⟵ α E_ν
3  update the node ν in the heap

SET_ERROR*(c, ν, v)
1  E_ν ⟵ v
2  C_ν ⟵ c
3  insert the node ν into the heap

DEC_ALL_ERROR*(β)
1  NOP()

LARGEST_ERROR*(c)
1  q ⟵ top node from the heap
2  f ⟵ arg max_{n∈N_q} E_n
3  return q, f

FIX_ERROR(c, ν)
1  E_ν ⟵ β^{λ(c−C_ν)} E_ν
2  C_ν ⟵ c
```

As can be seen, decreasing of all error variables is not necessary if each node keeps track of the cycle in which its error variable has been changed last time. The actual error, including its periodical decreasing, can be easily computed by Eq. (10) at any time.

Notice that Eq. (10) can be divided into two parts. The first part, $\beta^{(c_j-c_0)\lambda}E_0$, deals with the correction of the error due to its periodical decreasing. Let C_ν denote the cycle counter of the node ν. A new function FIX_ERROR fixes the error variable of the node according to the difference between the value of the current cycle c and the node's cycle counter C_ν, which also corresponds to the last call of FIX_ERROR.

The second part of Eq. (10), the sum term, represents the actual increasing of the node's error (INC_ERROR function). Notice the simple fact that each increasing of an error does not depend on any of the previous error additions. It depends entirely on a value of the step counter s (β and λ are constant parameters). This means that once the error is fixed to correspond with the current cycle, the change can be applied to the error regardless of any previous changes. This observation leads to the replacement of the original function INC_ERROR by a new function INC_ERROR*, which just fixes the error and increases the error (the third line of INC_ERROR* is explained in the next section). Similarly functions DEC_ERROR and SET_ERROR can be replaced by functions DEC_ERROR* and SET_ERROR*, respectively. All new functions are depicted in Algorithm 4.

5.2. Growing Phase

In the "growing" phase, the most time consuming part is the searching for the node with the largest error. An intuitive approach to speed up this operation would probably be to keep track of the node with the largest error and update it every time an error is changed. However, once this node is taken and its value is decreased, all nodes must be checked again to find the largest one and start the tracking again. This leads to the algorithm with a linear complexity, which we would like to avoid.

A priority queue can be used to store all nodes providing the node with the largest error in a constant time. The update operations for the heap-based priority queues have usually $O(\log n)$ complexity, where n is the number of the nodes on the heap. However, standard approaches of updates cannot be used because the actual values of error variables are unknown. Therefore, we propose the so called *lazy heap* to efficiently deal with the issue.

The lazy heap can be based on virtually any heap structure (the pairing heap [26] is used in our implementation) that have these four operations:

- *insert* - inserts a new node into the heap,
- *remove* - removes a node from the heap,
- *update* - updates a node's position in the heap when the error variable is changed,
- *top* - returns the node with the largest error (the node from the top of the heap).

Beside the selected underlying heap structure, the lazy heap utilizes an extra list L of nodes that are waiting to be inserted in the heap.

The insert operation does not insert the node into the heap but instead postpone this operation by inserting the node into the list L. The update operation works similarly. Instead of updating node's position in the heap, it completely removes the node from the heap and adds it to L. Finally, the most of the work remains for the top operation, which is why we call the structure lazy heap: all work is lazily postponed to the last moment when it is required.

The top operation works in two steps. In the first step, all nodes added so far into the list L are removed from the list and inserted into the underlying heap (using its original mechanism). In the second step, the node μ from the top of the heap is taken and its cycle counter C_μ is checked if it equals to the current value of the global cycle counter c. If it does, the node has the largest error; thus, it is the correct return value of the top operation. If the counters differ, FIX_ERROR is called for the node μ, its position in the heap is updated, and the top operation is called again. The proof of the correctness of this procedure is following.

Before we start constructing the proof a heap structure must be defined. We define a heap specifically for the purpose of containing nodes ordered by their errors,

but sufficiently generally to pose minimal restrictions on the selection of the underlying heap (priority queue). The proof itself starts with two lemmas that lead to Theorem 3 providing the proof by contradiction of the correctness of the top operation.

Definition 1. A *heap* is a structure that have exactly one *top* node μ that satisfy the property

$$E_{\mu,C_\mu} \geq E_{\nu,C_\nu} \tag{11}$$

for each node ν in the heap.

Lemma 1. *If $C_\nu \leq c$ then the property $E_{\nu,c} \leq E_{\nu,C_\nu}$ holds for each node ν in the heap.*

Proof. According to Eq. (4), the error in the cycle c equals to

$$E_{\nu,c} = \beta^{(c-C_\nu)\lambda} E_{\nu,C_\nu} \tag{12}$$

and since $\beta < 1$, the error in the cycle c must be smaller than (or equal to) the error of the same node in the cycle C_ν.

Lemma 2. *The top operation ensures that the node μ on the top of the heap will eventually satisfy $C_\mu = c$.*

Proof. During the top operation, if $C_\mu \neq c$ then the error is fixed and the node is inserted back into the *heap* with $C_\mu = c$. Since there is a finite number of nodes in the *heap*, a node with $C_\mu = c$ will eventually emerge on the *top* of the *heap*.

Theorem 3. *Let μ denote the top node of the heap. If $C_\mu = c$ and $C_\nu \leq c$ then property $E_{\mu,c} \geq E_{\nu,c}$ holds for all nodes ν in the heap. Hence, the node μ is the node with the largest error.*

Proof. According to Definition 1 $E_{\mu,C_\mu} \geq E_{\nu,C_\nu}$ must hold, which for $C_\mu = c$ leads to $E_{\mu,c} \geq E_{\nu,C_\nu}$. Now the proof by contradiction is constructed. Let assume that

$$E_{\mu,c} < E_{\nu,c} \quad \text{and} \quad E_{\mu,c} \geq E_{\nu,C_\nu}. \tag{13}$$

This is equivalent to

$$E_{\nu,C_\nu} \leq E_{\mu,c} < E_{\nu,c}, \tag{14}$$

which implies

$$E_{\nu,C_\nu} < E_{\nu,c}. \tag{15}$$

Eq. (15) is a contradiction with Lemma 1. Theorem 3 with conjunction with Lemma 2 proves that the aforementioned procedure provides the node with the largest error.

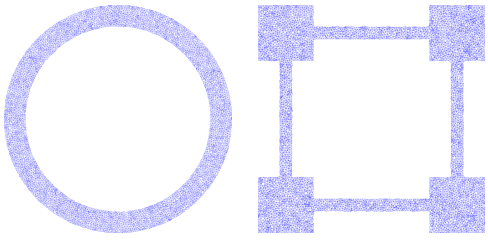
6. Experimental Results

The real benefit of the proposed optimizations has been evaluated in selected real problems to obtain a representative indicators of the improvements. The GNG has been tested with each optimization separately and with both optimizations combined in order to evaluate the impact of both optimizations proposed. The optimization of the nearest neighbor search has been compared with the vp-tree [23, 24] to show a performance of the well-known tree-based algorithm as well. The GNG has been tested in six variants:

- *Orig* – the original algorithm with the linear nearest neighbor (NN) search and without any modifications in the error handling,
- *VP* – the vp-tree used for the NN search and no error handling optimizations,
- *UG* – the uniform grid as proposed in Section 4 for the NN search and no error handling optimizations,
- *Err* – the linear NN search and error handling optimizations as proposed in Section 5,
- *VP+Err* - the vp-tree for the NN search and error handling optimized,
- *UG+Err* - the uniform grid for the NN search and error handling optimized – both of the proposed optimizations together.

Table 2: GNG parameters used in experiments

ϵ_b	ϵ_n	λ	β	α	A_{max}	h_t	h_ρ
0,05	0,0006	200	0,9995	0,95	200	0.1	1.5



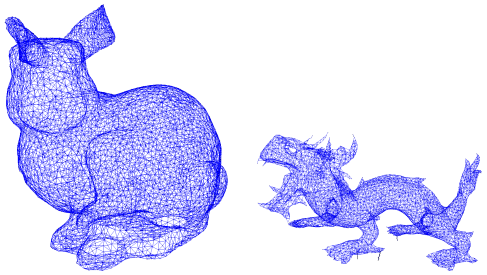
(a) circle (b) map

Figure 4: The GNG adapted to 2-D distributions.

Beside the GNG, the Growing Self-Reconstruction Maps (GSRM) [12] algorithm has also been evaluated because it is a nontrivial GNG based algorithm with an interesting application. The GSRM is an algorithm designed for a surface reconstruction of 3-D objects from a set of points. It adopts creation of nodes and edges from the GNG. In addition, it incorporates also a face creation between a triplet of edges. The resulting (reconstructed) surface is

then formed by a set of faces. The GSRM is evaluated using the same six variants as the GNG.

The evaluation of the tested GNG based algorithms has been performed using the following test cases: two 2-D distributions named *circle* (Fig. 4(a)) and *map* (Fig. 4(b)), and two 3-D distributions taken from Stanford repository [27], called *bunny* (Fig. 5(a)) and *asian-dragon* (Fig. 5(b)). The GSRM algorithm has been evaluated using the same *bunny* and *asian-dragon* datasets from Stanford repository.



(a) bunny (b) asian-dragon

Figure 5: The GNG adapted to 3-D distributions.

All algorithms have been implemented and evaluated within the same computational environment, i.e., a C implementation running at a GNU/Linux workstation with Intel Core i7 2.8 GHz processor. Therefore, the real benefits of the proposed optimizations can be directly compared using the real required computational time. In all tests performed, the stopping condition is the desired number of the nodes in the network. All parameters used are presented in Table 2.

Tables 3-6 show required computational times of the GNG algorithm computed as the median from five runs. The course of the time cost of the GNG depending on the number of nodes in the network is shown in Fig. 8 for the *circle* and in Fig. 9 for the *asian-dragon* distributions. Required computational times of the GSRM algorithm in the surface reconstruction of the *bunny* and *asian-dragon* datasets are presented in Tables 7 and 8, respectively, also computed as the median from five runs. The course of the time cost of 3-D reconstruction of the *asian-dragon* dataset depending on the number of nodes in the map up to 500,000 nodes is shown in Fig. 10. The result of the 3-D reconstruction with 50,000 nodes of the *asian-dragon* dataset is shown in Fig. 6.

The results indicate that the time cost of the variants *Orig*, *VP*, *UG* and *Err* grow exponentially with an increasing number of nodes in the network. The optimized variant with the vp-tree used for the NN search (*VP*) is faster than the original algorithm in all test cases but the efficiency of the optimization decreases with an increasing number of nodes. The reason is that with an increasing number of nodes the depth of the tree also increases, and the search query (together with the update operations) becomes more expensive. On the other hand, the variant with the uniform grid used for the NN search (*UG*) is in

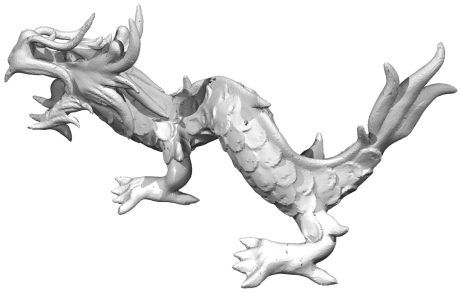
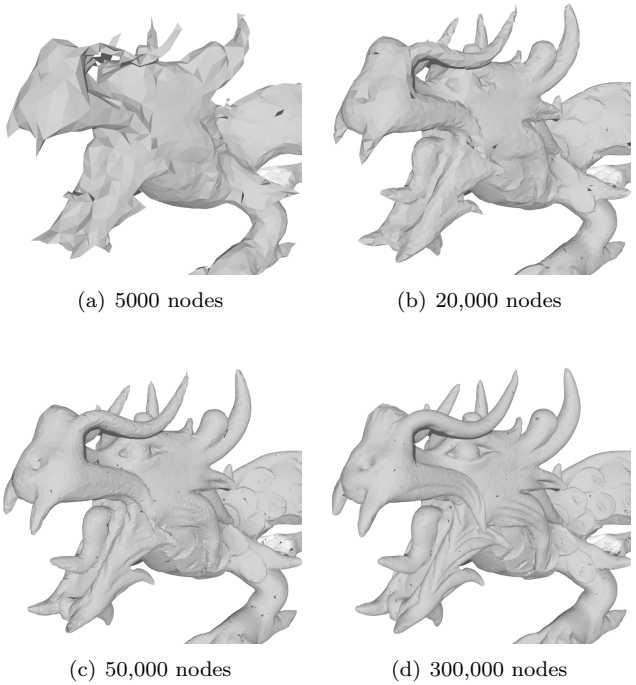


Figure 6: *Asian-dragon* reconstructed with 50,000 nodes by GSRM.

most cases almost 2 times faster than the *Orig* variant, more than 1.2 times faster than the *VP* variant, and the rate holds relatively stable with an increasing number of nodes in the network.

The variant with the optimized error handling (*Err*) is faster than the *UG* variant and the time cost increases more slowly. These results indicate that the error handling is the most time consuming part of the algorithm, even more than the nearest neighbor search.

Both the *VP+Err* and *UG+Err* variants speed up the original algorithms by several orders of magnitude. It can also be seen that the time cost grows approximately linearly in contrast to the exponential growth in other cases. Moreover, Fig. 10 indicates that the linear characteristics holds even for networks counting hundreds of thousands nodes. The *UG+Err* variant implementing both improve-



(a) 5000 nodes (b) 20,000 nodes

(c) 50,000 nodes (d) 300,000 nodes

Figure 7: Detail of reconstructed *asian-dragon* in different resolutions.

Table 3: Runtime of the GNG for the *circle* distribution

`circle`	1000	5000	10,000	25,000	50,000
Orig	1.27	46.35	190.41	1258.81	8294.14
VP	0.89	30.39	121.27	805.61	7018.69
UG	0.53	23.95	99.00	653.95	4813.62
Err	0.86	23.35	93.01	590.77	3418.27
VP+Err	0.47	3.32	7.42	21.19	48.04
UG+Err	**0.12**	**0.77**	**1.68**	**4.59**	**10.67**

Table 4: Runtime of the GNG for the *map* distribution

`map`	1000	5000	10,000	25,000	50,000
Orig	1.31	46.13	189.98	1228.15	7925.99
VP	0.94	30.72	121.73	804.20	6893.67
UG	0.52	23.84	98.98	657.47	4569.74
Err	0.89	23.49	94.26	602.91	3341.88
VP+Err	0.48	3.31	7.39	21.06	47.31
UG+Err	**0.12**	**0.78**	**1.67**	**4.60**	**10.80**

Table 5: Runtime of the GNG for the *bunny* distribution

`bunny`	1000	5000	10,000	25,000	50,000
Orig	1.42	49.62	202.04	1350.22	8628.54
VP	1.11	32.48	126.28	870.10	7226.30
UG	0.73	26.05	105.54	720.18	5136.57
Err	1.00	25.51	100.67	640.28	3466.06
VP+Err	0.63	3.87	8.41	23.16	50.58
UG+Err	**0.32**	**1.44**	**2.92**	**7.79**	**17.87**

Table 6: Runtime of the GNG for the *asian-dragon* distribution

`a-dragon`	1000	5000	10,000	25,000	50,000
Orig	1.40	61.93	292.44	1986.41	10,346.77
VP	1.04	46.98	210.90	1478.93	8,828.01
UG	0.68	37.46	185.47	1341.74	7,079.62
Err	0.98	25.04	99.23	626.36	3,374.74
VP+Err	0.62	3.90	8.49	23.50	51.81
UG+Err	**0.26**	**1.43**	**3.05**	**8.39**	**19.30**

Table 7: Runtime of the GSRM for the *bunny* dataset

`3d-bunny`	1000	5000	10,000	25,000	50,000
Orig	2.18	63.49	260.27	1837.71	13,446.30
VP	1.14	33.90	133.93	988.98	8,218.67
UG	0.81	29.03	118.56	829.09	6,591.20
Err	1.34	36.26	147.43	985.64	6,353.51
VP+Err	0.70	4.38	9.44	26.11	58.39
UG+Err	**0.40**	**1.94**	**3.96**	**10.64**	**24.59**

Table 8: Runtime of the GSRM for the *asian-dragon* dataset

`3d-dragon`	1000	5000	10,000	25,000	50,000
Orig	1.78	71.74	338.51	2314.30	14,817.25
VP	1.18	49.57	215.02	1430.08	9,610.59
UG	0.77	40.43	202.60	1359.61	8,149.79
Err	1.34	35.68	144.56	1006.85	6,012.06
VP+Err	0.69	4.37	9.50	26.75	60.02
UG+Err	**0.33**	**1.89**	**4.02**	**11.32**	**27.04**

ments proposed outperforms the variant with the vp-tree. The proposed techniques make the GNG (and GSRM) algorithm more than twice as faster as the *VP+Err* variant in all test cases.

7. Discussion

The exponential growth of the original GNG algorithm could prevent it from a deployment on problems, where a large size of the neural network is required. An example of the required high number of nodes is clearly visible in the case of the surface reconstruction using the GSRM algorithm. For the *asian-dragon* object, a reasonable resolution (where details of the object are distinguishable) starts with a relatively high number of nodes in the network (about 50,000) and with increasing size of the network a more precise reconstruction can be obtained (see Fig. 7). The original GSRM algorithm needs hours to reconstruct the object in a reasonable resolution and even days if a more precise reconstruction (e.g., with 300,000 nodes or more) is required, which practically disqualify the neural network algorithm from this sort of application.

Using the proposed speedup techniques, the reconstructed *asian-dragon* object with 50,000 nodes is provided within approximately 27 seconds, in contrast to more than three and half hours when the original algorithm is used. A resolution of 300,000 nodes is reached by the optimized GSRM within 7 minutes and resolution of 500,000 nodes within 14 minutes. The original GSRM was not able to reconstruct the object with 300,000 nodes even after 12 hours when we stopped the unfinished reconstruction containing 71,000 nodes. The results indicate that the proposed optimizations make the GNG algorithm usable for large scale problems.

The error handling optimization (*Err* variant) is the fastest particular optimization from all aforementioned techniques. Moreover, it modifies solely the internal mechanism of the error handling and does not depend on any externalities (such as chosen distance metric in the case of the NN search). Therefore, it is highly recommended to use this optimization even in cases where it is not possible to use any optimization of the NN search.

The combination of both proposed optimizations provide substantial speedup in all test cases and outperform all other tested variants of the algorithm. So, if the nearest neighbor search is performed in Euclidean space (with L_2 norm), it is recommended to integrate not only the error handling optimization, but also the proposed growing uniform grid.

The proposed optimization techniques are general and can be used for a high dimensional input space, where they can also provide a speedup. Our preliminary evaluations of the proposed techniques on high dimensional data indicate that the uniform grid does not provide a real benefit be-

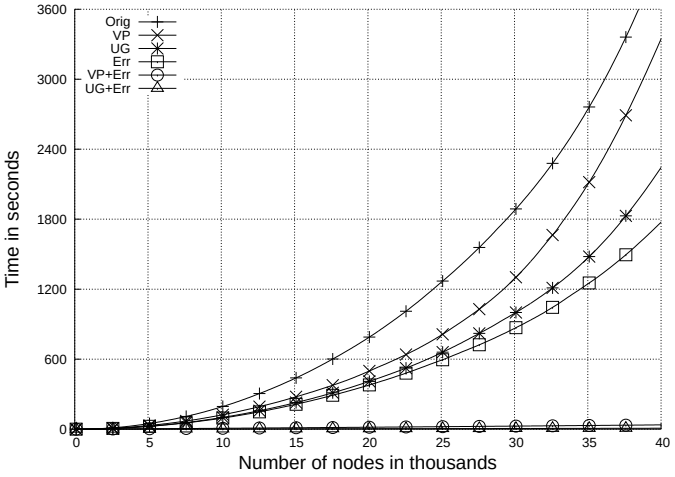


Figure 8: Required computational time of the variants of the GNG algorithm for the *circle* distribution.

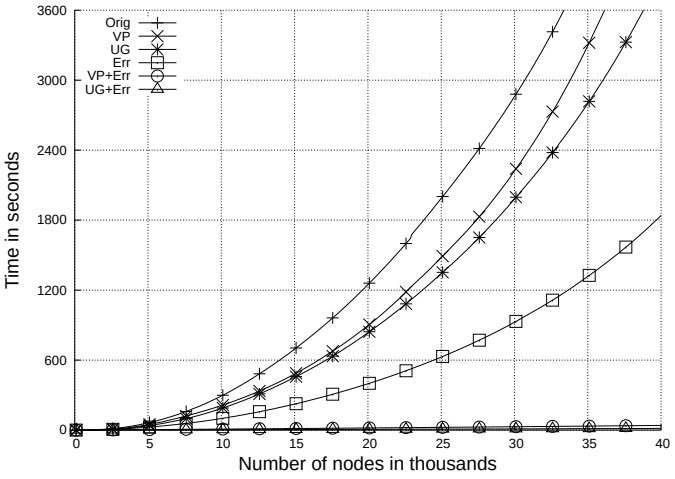


Figure 9: Required computational time of the variants of the GNG algorithm for the *bunny* distribution.

cause a number of cells that must be searched grows exponentially; thus, it degrades the performance of the uniform grid to the level of the linear search, and therefore, some tree like structure (e.g., vp-tree) should be rather used. On the other hand, the preliminary results indicate that the proposed error handling technique provides a substantial speedup (in several orders of magnitude) even on a high dimensional data. Although the real speedup improvements on high dimensional data can vary on the particular problems and representative performance indicators should be based on a large dataset of real problems, the results are very promising, and therefore, the presented optimization techniques (especially the proposed error handling) should be considered whenever the real time performance is an issue.

8. Conclusion

Two speedup techniques of the Growing Neural Gas (GNG) algorithm have been proposed. The first technique

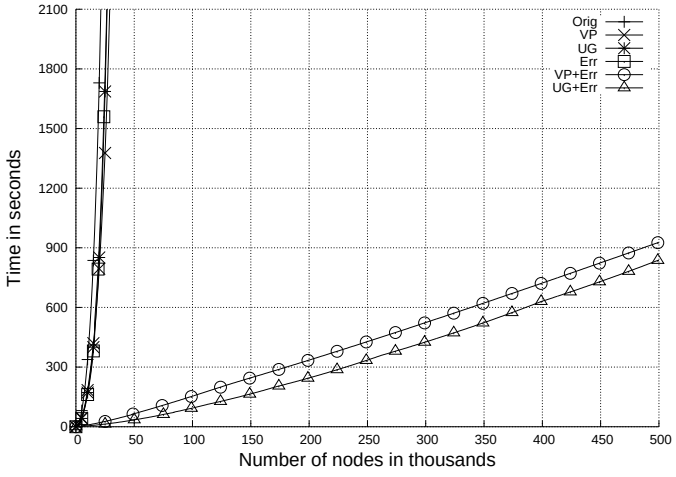


Figure 10: Required computational time of the variants of the GSRM algorithm for the 3D reconstruction of *asian-dragon* up to 500,000 nodes.

enhances the nearest neighbor search using a space partitioning by a grid of rectangular cells, which enables to update nodes in a constant time and reduces a search time due to reduction of the size of the space searched. The second technique speeds up the handling of node errors using the lazy evaluation approach.

It has been proven that the techniques do not change whatsoever the effective behavior of the algorithm. The experimental results indicate that both techniques separately can provide a considerable speedup. If an optimization technique for the nearest neighbor search (the proposed UG or vp-tree) is combined with the proposed error handling optimization then the GNG and GNG based algorithms run faster by several orders of magnitude and the time cost grows approximately linearly with an increasing number of nodes in the network. A huge performance boost has been demonstrated for 2D and 3D problems; however, the presented techniques are general and works also for high dimensional input spaces, where a similar speedup is expected. A detailed performance evaluation of high dimensional problems is a subject of our further work.

9. Acknowledgment

This work has been supported by the Technology Agency of the Czech Republic under Project No. TE01020197 and by the Ministry of Education of the Czech Republic under Project No. LH11053.

Appendix A. Software

The source code of the optimized GNG and GSRM algorithms is part of the *fermat* library [28] publicly available under 3-clause BSD license.

References

[1] B. Fritzke, A growing neural gas network learns topologies, in: G. Tesauro, D. S. Touretzky, T. K. Leen (Eds.), Advances in Neural Information Processing Systems 7, MIT Press, Cambridge MA, 625–632, 1995.

[2] T. Kohonen, Self-organized formation of topologically correct feature maps, Biological Cybernetics 43 (1) (1982) 59–69.

[3] B. Fritzke, Growing Cell Structures—a Self-Organizing Network in k Dimensions, in: I. Aleksander, J. Taylor (Eds.), Artificial Neural Networks, vol. II, North-Holland, Amsterdam, Netherlands, 1051–1056, 1992.

[4] B. Fritzke, Growing Grid - a self-organizing network with constant neighborhood range and adaptation strength, Neural Processing Letters 2 (1995) 9–13.

[5] T. Kohonen, E. Oja, O. Simula, A. Visa, J. Kangas, Engineering applications of the self-organizing map, Proceedings of the IEEE 84 (10) (2002) 1358–1384.

[6] D. Deng, N. Kasabov, On-line pattern analysis by evolving self-organizing maps, Neurocomputing 51 (2003) 87 – 103.

[7] H. Frezza-Buet, Following non-stationary distributions by controlling the vector quantization accuracy of a growing neural gas network, Neurocomputing 71 (7-9) (2008) 1191 – 1202.

[8] K. Doherty, R. Adams, K. A. J. Doherty, R. G. Adams, N. Davey, Hierarchical Growing Neural Gas, in: Proc. Int. Conf. Adaptive and Natural Computing Algorithms, 140–143, 2005.

[9] V. J. Hodge, J. Austin, Hierarchical Growing Cell Structures: TreeGCS, IEEE Trans. Knowledge and Data Engineering 13 (2000) 2001.

[10] A. Qin, P. Suganthan, Robust growing neural gas algorithm with application in cluster analysis, Neural Networks 17 (8-9) (2004) 1135 – 1148.

[11] D. X. Le, G. R. Thoma, H. Wechsler, Document classification using connectionist models, in: 1994 IEEE International Conference on Neural Networks. IEEE World Congress on Computational Intelligence, vol. 5, 3009–14, 1994.

[12] R. L. M. E. Do Rêgo, A. F. R. Araújo, F. B. De Lima Neto, Growing Self-Reconstruction Maps, IEEE Transactions on Neural Networks 21 (2010) 211–223.

[13] V. L. D. Mole, A. F. R. Araújo, Growing self-organizing surface map: Learning a surface topology from a point cloud, Neural Computation 22 (2010) 689–729.

[14] J. Barhak, Reconstruction of Freeform Objects with Arbitrary Topology from Multi Range Images, Ph.D. thesis, Mechanical Engineering faculty, Techion - Israel institute of technology, Haifa, Israel, 2003.

[15] I. Ivrissimtzis, W.-K. Jeong, S. Lee, Y. Lee, , H.-P. Seidel, Neural meshes: surface reconstruction with a learning algorithm, Research Report MPI-I-2004-4-005, Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany, 2004.

[16] Y. Prudent, A. Ennaji, An incremental growing neural gas learns topologies, in: IJCNN, vol. 2, IEEE, 1211–1216, 2005.

[17] A. Ocsa, C. Bedregal, E. Cuadros-Vargas, DB-GNG: A constructive Self-Organizing Map based on densilty, in: IJCNN, IEEE, 1953–1958, 2007.

[18] M. Saska, V. Vonásek, M. Kulich, D. Fišer, T. Krajník, L. Přeučil, Bringing reality to evolution of modular robots: bio-inspired techniques for building a simulation environment in the SYMBRION project, in: IROS 2011, Workshop on Reconfigurable Modular Robotics: Challenges of Mechatronic and Bio-Chemo-Hybrid Systems, 2011.

[19] C. Traina, A. Traina, B. Seeger, C. Faloutsos, Slim-trees: High performance metric trees minimizing overlap between nodes, in: 7th International Conference on Extending Database Technology (EDBT), Springer-Verlag, 51–65, 2000.

[20] M. R. Vieira, C. Traina, F. J. T. Chino, A. J. M. Traina, DBM-Tree: A Dynamic Metric Access Method Sensitive to Local Density Data, in: SBBD, 163–177, 2004.

[21] A. Guttman, R-trees: A Dynamic Index Structure for Spatial Searching, in: International Conference on Management of Data, ACM, 47–57, 1984.

[22] M. De Berg, O. Cheong, M. van Kreveld, Computational geometry: algorithms and applications, Springer, 2008.

[23] P. N. Yianilos, Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces, in: Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 1993.

[24] A. W.-c. Fu, P. M.-s. Chan, Y.-L. Cheung, Y. S. Moon, Dynamic vp-tree indexing for n-nearest neighbor search given pair-wise distances, The VLDB Journal 9 (2000) 154–173.

[25] M. Edahiro, I. Kokubo, T. Asano, A new point-location algorithm and its practical efficiency: comparison with existing algorithms, ACM Trans. Graph. 3 (2) (1984) 86–109.

[26] M. L. Fredman, R. Sedgewick, D. D. Sleator, R. E. Tarjan, The pairing heap: a new form of self-adjusting heap, Algorithmica 1 (1986) 111–129.

[27] The Stanford 3D Scanning Repository, World Wide Web electronic publication, URL `http://www-graphics.stanford.edu/data/3Dscanrep/`, 2009-02-12.

[28] Fermat Library, World Wide Web electronic publication, URL `http://www.danfis.cz`, 2011-08-30.