Jussi Pakkanen, Jukka Iivarinen, and Erkki Oja, The Evolving Tree – Analysis and Applications. IEEE Transactions on Neural Networks, vol. 17, number 3, pages 591-603, May 2006.

The Evolving Tree — Analysis and Applications

Jussi Pakkanen, Jukka Iivarinen, and Erkki Oja, *Fellow, IEEE*

***Abstract*— In this paper we enhance and analyze the *Evolving Tree* data analysis algorithm. The suggested improvements aim to make the system perform better while still maintaining the simple nature of the basic algorithm. We also examine the system's behaviour with many different kinds of tests, measurements and visualizations. We also compare the Evolving Tree's performance against classical data analysis methods and very similar modern systems. We find that the Evolving Tree is a suitable method for unsupervised analysis of huge data sets.**

***Index Terms*— hierarchical clustering, Evolving Tree, tree-shaped neural networks, self-organization**

I. Introduction

The Self-Organizing Map (SOM) is a widely used tool in various data analysis tasks [1]. However, some of its intrinsic features make it unsuitable for analyzing very large scale problems. Almost all operations on SOM start by locating the best matching unit (BMU) among all the nodes. This scales linearly according to the map size. Analyzing huge data sets requires very large maps. While operating on these maps is not usually intractable, it can be extremely slow. Another drawback is that the map size must be chosen beforehand. While there are some heuristics for this, experimenting with different sized maps is quite time-consuming.

There have been several different approaches to solve these problems. They can be roughly divided into two different groups. The first ones are flat systems that grow during training. Examples include incremental grid growing [2], growing cell structures [3], and dynamic cell structure [4]. The second group tries to build efficient search structures to make operations faster. This is a very common problem in various fields, which has resulted in a plethora of different algorithms [5]. Most of these algorithms are based on classical computer science rather than neural computation. An exception is the tree-structured SOM [6].

An efficient way of attacking large scale problems is called *divide and conquer*. This approach divides the problem into smaller subproblems, which are easier to solve and combine. In euclidean spaces this usually means partitioning the space hierarchically into smaller and smaller regions. This partitioning is usually presented as a search tree, which is a traditional and efficient way of doing searches. Another advantage of divide and conquer is an easy approach to complexity control. When the space is partitioned, the subspaces can be further partitioned without affecting other areas of the data space. This converts complexity control from a global problem to a local one which is simpler.

There are some hybrid systems that try to combine both of these features. Even though they start from relatively similar premises, the resulting systems vary quite a lot. Some systems use the hierarchic structure only as an aid in classification, like TreeGCS [7]. Others, like SAINT [8], DGSOT [9], GHSOM [10] and CNeT [11], use the hierarchical structure to speed up training and queries.

In this paper, we present and analyze the Evolving Tree [12], [13], [14] (ETree) and its behaviour. The Evolving Tree is a new kind of self-organizing neural network that has been designed to scale to very large problems.

At its basic level the Evolving Tree is a fast, hierarchical clustering method especially suitable for large, high-dimensional problems. This is not the only way of seeing the algorithm. It can also be seen as an index to vectorial data, similarly to R-tree [15] and other such algorithms. The problem with these approaches is that they become exponentially slower as the data dimension increases [5]. The hierarchical and approximate nature of ETree allows it to tackle problems that are too large for these classical methods.

Contents of the paper are as follows. In chapter 2 we give an overview of the Evolving Tree's algorithms and architecture. This is followed by the novel improvements and analysis in chapter 3. In chapter 4 we subject our system to several qualitative and quantitative experiments. Then we discuss the results and conclude the paper.

II. The Evolving Tree

We will now briefly describe the basic Evolving Tree algorithm. A more detailed description of the basic algorithm can be found in [14]. We start with the small Evolving Tree in Figure 1 (a). This example tree has a fanout of 2 for simplicity. In practice larger values are often used. The tree consists of black *leaf nodes* and white *trunk nodes*. Each node has a *prototype vector* w_i, which places it somewhere in the data space. It also has a counter b_i, which tells how many times it has been the best matching unit. Assume that we have available a training set of data vectors, which are used in training one by one. Training the tree starts by finding the best matching unit (BMU) with a greedy tree search. For every training vector x_i we start at the root node and select the child which is closest to the training vector. We select this node and examine its childen and so on until we reach a leaf node. This is the BMU.

Now we need to update the leaf node locations. We use the Kohonen learning rule [1]:

$$\mathbf{w}_i(t+1) = \mathbf{w}_i(t) + h_{ci}(t)[\mathbf{x}(t) - \mathbf{w}_i(t)]. \quad (1)$$

The function h_{ci} defines the amount of adaptation and is a gaussian function as in SOM:

$$h_{ci}(t) = \alpha(t) \exp\left(\frac{d(r_c, r_i)^2}{2\sigma^2(t)}\right). \quad (2)$$

The authors are with the Neural Networks Research Centre, Helsinki University of Technology, 02015 HUT, Finland (email: jussi.pakkanen@hut.fi)

Here α and σ are used to control the width and time decay of the neighborhood function. They are usually exponentially decreasing functions with respect to time.

The problematic part is the function $d(r_c, r_i)$, which tells how far apart the nodes r_c and r_i are in the SOM grid. The Evolving Tree does not form a grid so some other method is required. We use an equivalent metric called the *tree distance* which can be seen in Figure 1 (b). The idea is to calculate the amount of "hops" needed to get from one node to the other along the tree. In this case five hops are needed to get from A to B. Using this distance the SOM neighborhood function can be applied.

These two steps — finding the BMU and updating leaf nodes — form most of the training. The third step is growing the tree. Every node has a counter that tells how many times it has been the BMU. When the counter reaches a certain value, called the *splitting threshold*, the node is split. That is, it is given some child nodes, thus becoming a trunk node. The child nodes' prototype vectors are initialized to their parent's value. The training may now continue using the new, slightly larger tree. One of the consequences of these training rules is that different branches have larger tree distances than nodes in a same branch. This is expected since different branches grow to different areas in the data space.

Now we can describe the very simple basic ETree algorithm for a single training vector.

1) Find the BMU using the search tree.
2) Update the leaf node locations using the SOM training formulas substituting tree distance for grid distance.
3) Increment the BMU's hit counter.
4) If the counter reaches the splitting threshold, split the node.

This is repeated for every vector on the training set until the system is deemed good enough. Usually this means going through the data a pre-specified amount of times. Because distances between branches are usually quite large, the weight updates become very small. The computational load can thus be reduced by only updating the nodes close to the BMU. These can be efficiently found using the links between nodes.

III. Enhancements and analysis of the Evolving Tree

In this paper we further examine the established Evolving Tree algorithm. First we consider different ways of analyzing and enhancing the basic algorithm. Then we derive the computational complexity of the algorithm including the enhancements. The effects of the enhancements are studied experimentally in the next chapter.

A. Controlling the growth

A very important part of the training process is determining how large the tree should become. If there are too few nodes, we only get a very coarse overview of the data space. Too many nodes may cause some overfitting but even more seriously they add unnecessary computational burden. Therefore it is important to have an efficient stopping criterion.

Approaches based on e.g. mean quantization error and average distortion have been applied in related works [9], [10]. These approaches have a solid mathematical background, but there are also drawbacks. The biggest one is that these distortion functions must be calculated relatively often. This can slow down the training considerably.

We propose an algorithm that is based on the concepts of regularization and weight decay [16]. The basic concept of regularization is to penalize overly complicated models, thus preventing overfitting. One approach is to create a penalty function and to add that to the objective function to be minimized. An example penalty term is a sum of all the weights of a multilayer perceptron. This term can be made smaller by decaying all the weights toward zero between epochs.

In our system we would gain nothing by decaying the prototype vectors. That is because the locations of the nodes are not the cause of overfitting. The real reason is too many nodes. To restrict the size of the tree, we apply weight decay to the BMU counters b_i. This yields the following two-phase algorithm, which both controls the size of the tree and automatically stops the training. The results are examined in subsection IV-A.

1) At the end of each epoch set $b_i(t+1) = \gamma \cdot b_i(t)$ for all nodes i, where $\gamma \in]0, 1[$.
2) If the tree has grown less than a specified amount (e.g. 5%) since the last epoch, stop training.

B. Removing layers

One beneficial feature of most neural networks is *graceful degradation*. If the system suffers some sort of a minor damage, its effects on the total operation should be small. An example is randomly distorting a weight of one node in a multilayer perceptron. The total performance may suffer slightly, but the system is still operational and useful.

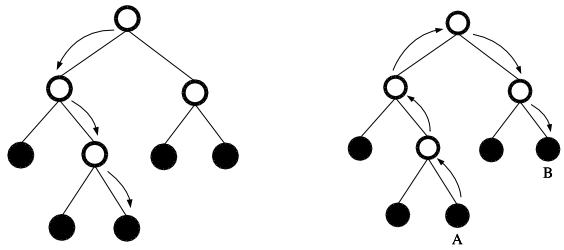
To examine whether the Evolving Tree also possesses this property we examined its performance when nodes are removed. The results can be found in subsection IV-B.

C. Better search for the BMU

One source of error in the Evolving Tree may be the greedy BMU search, which is not guaranteed to find the globally optimal node. We examined alternate ways of finding the BMU. The first one is the original algorithm with a slight modification. At every layer we keep the n best subbranches instead of only one. When we reach the leaf nodes the best one is selected as the BMU. The second algorithm just finds the global BMU among leaf nodes, ignoring the tree altogether except for the leaf node updates. This algorithm can be seen as a growing variant of k-means clustering. The results can be found in subsection IV-C.

D. Child node initialization

When new child nodes are created, they are placed on top of their parent in the data space. As the training progresses



(a) BMU search (b) Tree distance

Fig. 1. Fundamental operations of the Evolving Tree.

the children move to different areas in the data space. Using a different initialization scheme could lead to improved computational efficiency and faster convergence.

We examined two initialization schemes. The first perturbs the child nodes randomly. The second one is based on principal component analysis (PCA). The idea is that when a node is split, we first find all data vectors that map to it. Then we calculate the principal components of this data cloud. We then take the plane spanned by the two largest principal components and the node to be split. The child nodes are placed evenly on an ellipse that lies in this plane. The center of the ellipse is the parent node and the ellipse axes are the principal component vectors. The axis lengths are the standard deviations of the respective components.

The basic idea of this scheme is to spread out the child nodes to the directions of the largest variances. In the case of two children this scheme reduces to placing the nodes along the axis of the first principal component. In the case of four children they are placed along the two first principal components. The benefit of the ellipse algorithm is that we can have an arbitrary amount of child nodes and still spread them out efficiently. The efficiency of these algorithms is examined in subsection IV-D.

E. Optimizing the leaf node locations

A known property of the SOM is that if the neighborhood function does not go to zero the resulting map will be nonoptimal with respect to quantization error. Since the amount of training epochs is decided beforehand, the rate of decrease in SOM can be chosen properly. When training the Evolving Tree using weight decay, the amount of epochs is unknown. This suggests that the algorithm might not find optimal locations for the leaf nodes.

One way of optimizing the locations, while still maintaining the system's unsupervised nature, is to use a variant of k-means clustering. First we map all training vectors to leaf nodes using the established BMU search. Then we move the leaf nodes to the center of mass of their respective data vectors. This procedure is repeated a few times to obtain the final leaf node locations. Experimental results are given in subsection IV-E.

Regular k-means usually requires several iterations to obtain good results. One of the reasons is that initializing it effectively is somewhat tricky. In our case the tree has partitioned and initialized the data space quite effectively so only a few rounds are needed. It should also be noted that the Evolving Tree could be utilized as a fast initialization step for regular k-means clustering. The drawback is that the search tree must then be discarded, losing both the neighborhood information and the ability to do fast queries.

F. Computational complexity

Let us use the following notation.

d	data space dimension
N	data set size
h	tree depth
k	# of updated neighbours
b	branching factor (fanout)
θ	splitting threshold
l	# of leaf nodes

The computational complexity is the sum of finding the best matching unit (C_{BMU}) and updating the neighbours (C_{ud}). The complexity of one epoch is simply the complexity of a single operation multiplied by the epoch size.

To find the BMU we have to calculate a vector distance b times at every level of the tree. The amount of calculations needed is

$$C_{BMU} = d \cdot h \cdot b. \tag{3}$$

Updating the leaf locations means moving k nodes. We also have to add the cost of splitting the leaf nodes. This happens, on average, every $1/\theta$ steps, making the total cost of the update for a single vector

$$C_{ud} = k \cdot d + \frac{b}{\theta} \tag{4}$$

Adding these and multiplying by the epoch size N we find that the amount of calculations needed for a single epoch is

$$\begin{aligned} C_{tot} &= N\left(C_{BMU} + C_{ud}\right) &(5)\\ &= N\left(d \cdot h \cdot b + k \cdot d + \frac{b}{\theta}\right). &(6) \end{aligned}$$

We can see that the complexity is linear in data dimension d. This is very desirable, as many methods are exponential as mentioned in the introduction. To find the complexity with respect to the data set size N, we first note that d, b and θ are constants, so they are dropped. Because a search tree is formed during training, it follows that $h \propto \log N$. The most difficult parameter to define exactly is the amount of updated neighbors k. In our experiments we have found that only a couple of nodes are updated. This is due to the rather narrow neighborhood functions and large intra-branch distances. Therefore we can approximate that k is a constant and the computational complexity for one epoch reduces to

$$O(N \log N) \tag{7}$$

1) Complexity of weight decay: Weight decay is a very light operation, which is run at most once per epoch. It affects every child node once, so the complexity is

$$C_{wd} = l \cdot d \tag{8}$$

To function effectively, every node models only a small portion of the data space. This means that only a few data vectors map to each leaf node. Therefore $l \propto N$ and the complexity per round is

$$O_{wd}(N). \tag{9}$$

2) Complexity of the k-means phase: Complexity of the k-means updating is quite simple to calculate. That part consists of mapping all data vectors to leaf nodes and then moving the leaf nodes to the center of mass of their respective vectors. For a single epoch that means

$$C_{km} = N(d \cdot h \cdot b) + N \cdot d. \tag{10}$$

Using the same reasoning as above we find that

$$O_{km}(N \log N). \tag{11}$$

The k-means portion is usually run only a few times at the end of training. In this case the computationally dominant portion is training, not the final k-means adjustment.

3) Total complexity: The largest complexity of any part of the algorithm is $N \log N$. Therefore the complexity of the Evolving Tree is also $N \log N$ per epoch. To find the total complexity, we have to determine the total number of epochs as a function of N. Unfortunately the tree shape and weight decay make determining it analytically extremely difficult. Larger databases need larger trees which indicate more epochs. On the other hand if the epoch is large, more splits occur per epoch. Therefore the tree grows more for large epochs than for small ones, indicating that less epochs are required.

Empirically we have discovered that relatively few epochs, ten to twenty, are needed, even for databases of twenty thousand vectors.

4) Complexity of k-means and SOM: We briefly analyze the complexity of regular k-means clustering and SOM, since they are used in experiments later. First we note that both SOM and k-means should have about the same amount of nodes as the ETree has leaf nodes, meaning $O(N)$. That is also the complexity for a single BMU search, since there is no search structure. One batch has N of these searches and therefore the total complexity of a single training round is $O(N^2)$.

IV. Experiments

Three kinds of experiments were performed. The first ones examine the effects of the enhancements A to E that were described previously and to examine how the algorithm behaves under various circumstances. Even though all experiments do not lead to improved performance, they are still useful in understanding the inner workings of the algorithm.

The second class of experiments compare ETree to classical data analysis methods. These experiments give us baseline performance values for classification and training time. For this reason we have selected algorithms that work in a roughly similar way as ETree. The main difference is that they are flat instead of hierarchical.

Finally we compare ETree to two other tree-shaped neural network systems, S-Tree and CNeT. These systems have a similar divisive, hierarchical approach to data analysis. These experiments show ETree's performance when compared to modern systems.

We have used two different kinds of data sets for the experiments. The first one consists of different MPEG-7 features [17] which have been calculated for 1300 paper surface defect images [18]. This gives us moderate sized databases with dimensions around 20. The exact dimension varies somewhat depending on the descriptor being used. This data set represents actual industrial data. There are a total of 14 different classes which are fuzzy and overlapping and therefore extremely difficult to classify.

We calculated the following MPEG-7 features for these defect images.

- *Color Structure (CS)* slides a structuring element over the image. The numbers of positions where the element contains each particular color are stored and used as a descriptor.
- *Edge Histogram (EH)* calculates the amount of vertical, horizontal, 45 degree, 135 degree and non-directional edges in 16 sub-images of the picture, resulting in a total of 80 histogram bins.
- *Homogeneous Texture (HT*) filters the image with a bank of orientation and scale tuned filters that are modeled using Gabor functions. The first and second moments of the energy in the frequency domain in the corresponding sub-bands are then used as the components of the texture descriptor.

The other data set consists of 18 000 handwritten digits. The digit images were normalized to 32×32 pixels with 256 gray scale values. 38 principal components were obtained using PCA and used for our experiments [19]. This is a relatively large classified database. Comparing results to the

previous databases tells us how different algorithms scale up. The experiments test relative diverse areas. It is beneficial to see how the algorithm performs under different circumstances.

A crucial question is the performance measure used in comparisons. We are mostly interested in unsupervised learning, so suitable measures are quantization error and clustering indices such as the Davies-Bouldin index. These comparisons can be found in IV-G. Visualization experiments are found in IV-F.

However a very feasible comparison is also clustering with ground truth available: how well do the clusters produced by the algorithms correspond to labeled classes found in the data. In order to carry out such classification experiments a voting rule must be used. All classification results have been obtained by using majority voting with ties broken arbitrarily and ten fold cross validation.

A. Effects of regularization

Effects of the regularization coefficient γ were examined by training the tree with different values and comparing the classification results. We used a very aggressive splitting scheme to amplify the differences. All experiments were done with the homogeneous texture data set. The results can be seen in Figure 2. The results follow the theory quite nicely. Tree size, which is measured as the total amount of nodes, depends very strongly on the coefficient, and smaller trees produce worse classification rates. This degradation is not strong but still very clear, but this is due to the very small splitting threshold, as most of the resulting nodes are somewhat redundant. We verified these results by measuring the average quantization error. We found that it increased from 0.12 to 0.16 as we decreased the regularization coefficient. The decrease had the same linear shape as the classification error. In practice we have found that values of γ between 0.85 and 0.95 produce quite good results.

B. Removing layers

For simplicity we removed whole layers at a time. We used the homogeneous texture data set. The tree was trained to be larger than usual so we would have more layers to remove.

The results in Figure 3 show that the Evolving Tree degrades very gracefully. The classification rate drops dramatically only after three layers are totally removed from the tree. After that the rate goes rapidly to zero. This makes the Evolving Tree very robust against small errors in its lowest layers.

C. Better search for the BMU

Figure 4 shows the differences between the regular BMU search and the two new algorithms. After training the BMUs were found using all the algorithms and the tree distances between these were calculated. The plots show the distribution of distances from the regular BMU to ones obtained with the alternative methods. These results are for the edge histogram database, but others gave very similar results.

The first thing we notice is that the regular algorithm finds the global BMU only approximately 46% of the time. The second observation is that there are no instances of distance one in either case. Nodes with a distance of one share a common parent. If the search algorithm reaches that parent, it will always select the correct BMU, since that is the locally optimal solution.

At first sight the distances seem small, but they are larger than they appear. The largest distance in the figure is 9, which corresponds to a subtree depth of 5. If we assume a splitting factor of 4, this subtree would have a maximum of $4^4 = 256$ leaf nodes. This is approximately the size of the whole tree used in these tests. The conclusion is that in some cases the greedy BMU search may choose the wrong branch at the very first step. While this seems quite bad at first sight, it is actually not. Two nodes that have a very large tree distance may still be very close to each other in the data space. This happens when the nodes are very close to a partition line set by the topmost nodes. This is an unfortunate side-effect of the hierarchical structure.

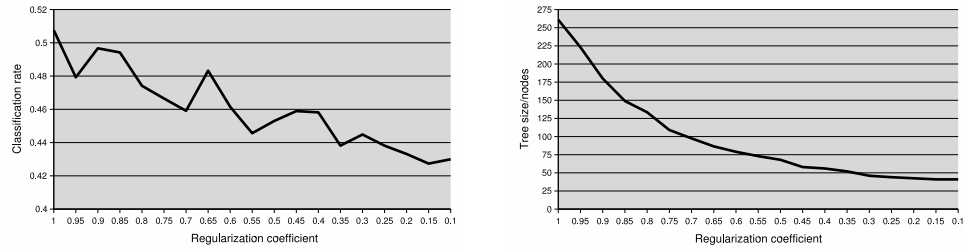
The important question is, however, whether the inoptimality in the search algorithm affects the classification rates. A partial answer can be found in Figure 5. It has the per-class classification percentages for the different algorithms. The relative performances fluctuate slightly for different classes. None of the methods is better for all the classes. This is caused by the overlapping classes, where improving the total rate happens at the expense of some classes. Averaged results are 51% for regular ETree, 52% when using two BMUs and 55% for the global BMU. The last rate approaches the result for regular k-means (Figure 7 (a)). This is understandable as the global BMU algorithm reduces to a variant of k-means as described in subsection III-C. Quantization error behaves in the same way with global BMU search having the smallest error.

The intuitive behaviour would be that if the system can not find the real BMU it should not be able to model the data properly. This seems not to be the case here. The explanation lies in the way the Evolving Tree handles the data space. Each leaf node explains only a very small portion of the data space. If a vector maps to a node, then very probably vectors in its immediate vicinity map to the same node. This happens whether the node is the global BMU or not. Obviously this property does not always hold, which is why using the global BMU gives better results.

These results indicate that there is no pressing need to replace the greedy BMU search with another algorithm. Using several BMUs gives only an insignificant performance boost. Using global BMU gives a bigger improvement. The drawback is that the search tree is ignored, so the computational complexity increases significantly. One of the design goals of the Evolving Tree has been scalability to very large problems, which makes the global BMU search unsuitable.

D. Child node initialization

The main results can be seen in Figure 6. Figure a lists the results for four different algorithms. The first bar is the regular ETree, the second bar uses the optimal BMU search described above. The third one uses the PCA initialization and the fourth one combines PCA and optimal search. The results



(a) Classification decay (b) Tree size decay

Fig. 2. Effect of regularization coefficient to performance, homogeneous texture data set.

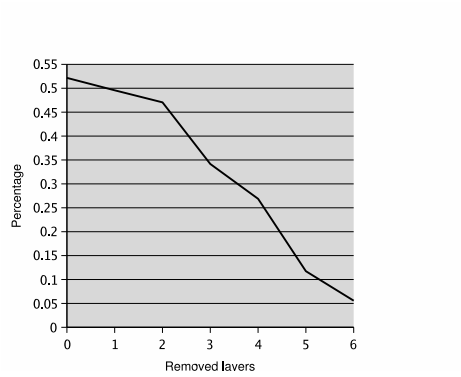
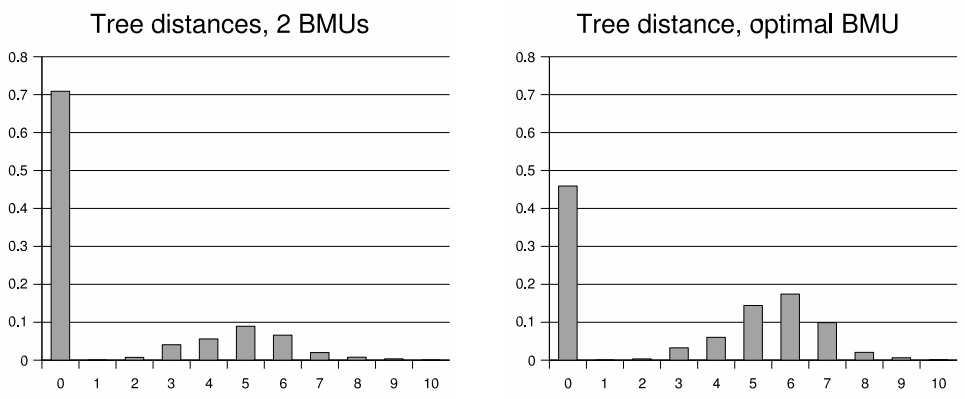


Fig. 3. Performance degradation when removing layers, homogeneous texture data set.



(a) (b)

Fig. 4. Tree distances to regular algorithm's BMU on the edge histogram database.

for random child initialization have been omitted because they are all but identical to the regular initialization scheme. This is expected, since the first couple of update rounds effectively move the nodes around in a pseudo-random fashion.

All the methods give almost identical classification results. Only edge histogram seems to benefit from the PCA initialization, but only a few percents. Interestingly the optimal BMU search does not give better results in all cases. This is most likely caused by the very difficult nature of the data set as discussed earlier. PCA's good performance on edge histogram is countered by the drop in classification rate in the other two cases. Overall the PCA initialization does not seem to produce noticeable improvements. A negative effect is that training times grow very large, because PCA is a computationally

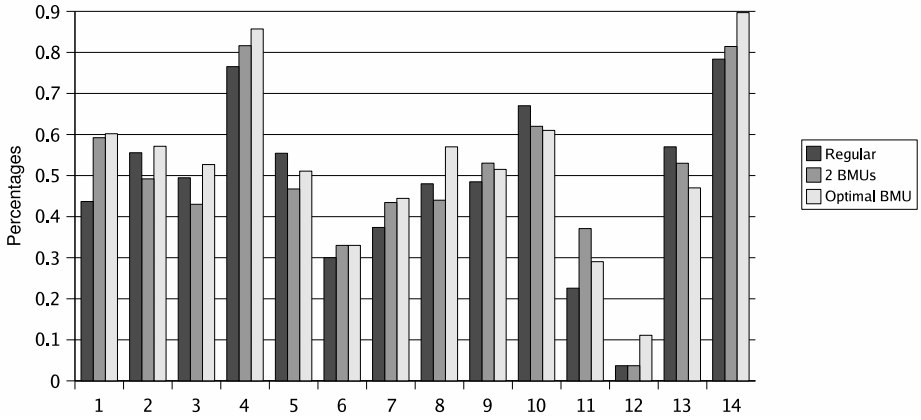
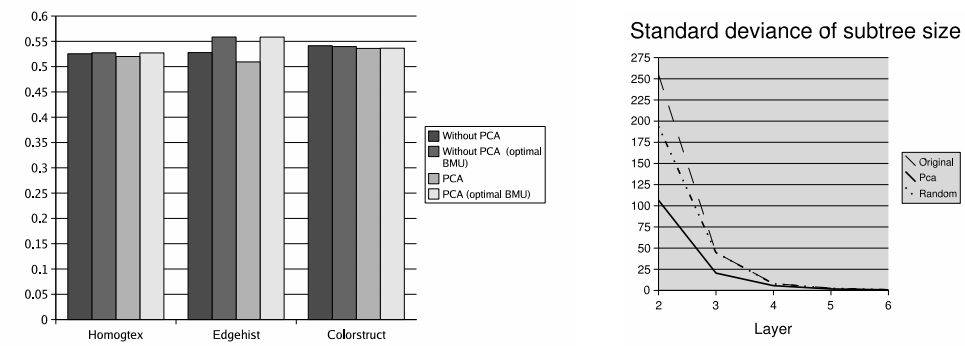


Fig. 5. Classification rates for the different BMU algorithms per class using edge histogram.



(a) Classification rate

(b) Subtree size

Fig. 6. Comparing different initialization schemes.

costly operation.

PCA initialization does have its benefits as can be seen in Figure 6 (b). It shows the standard deviation of subtree sizes for every layer in the Evolving Tree. This particular tree was grown with very aggressive splitting parameters, so the differences between algorithms can be seen more clearly. Smaller deviance is better because it implies a more balanced tree.

The random initialization tree is slightly more balanced than the regular tree. The difference is not very large, though. PCA produces the most balanced trees. The difference between different algorithms is very small at layer three and almost insignificant at layer four.

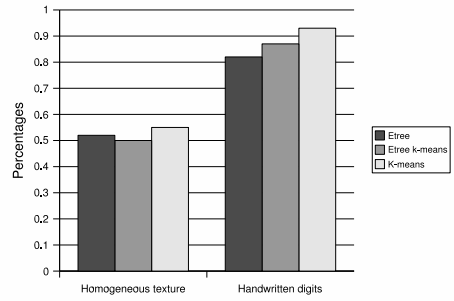
The results above suggest that PCA initialization could be used in the top layers if tree balance is absolutely vital. This is usually not the case since the regular ETree produces quite a good search tree. Another problem is that computing the principal components for a large data base is an expensive operation. Since the PCA initialization does not yield noticeably better qualification results, its computational complexity is not usually justifiable.

E. Comparison to K-means clustering

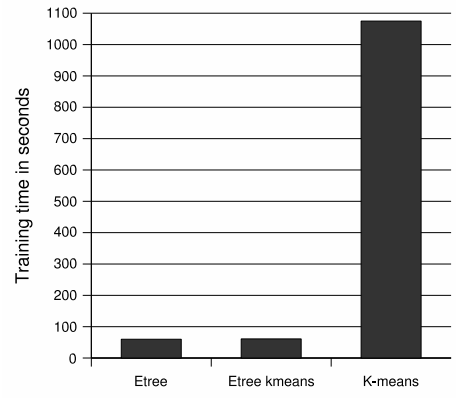
First we ran the classification tests on the Evolving Tree. After those tests we checked how many leaf nodes the trees had on average. Then we did the comparison tests with k-means using the same amount of clusters. The main results can be seen in Figure 7.

First in subfigure (a) we have classification results for the small database. In this case the classification rates are almost identical. Interestingly the k-means adjustment slightly worsens ETree's results. This indicates that with medium sized databases the algorithm optimizes the leaf node locations very well by itself, so the k-means adjustment is not needed. Another factor is most likely the fuzzy nature of the data. Results with other MPEG-7 data sets were very similar.

Performance on the larger database is very interesting. The regular ETree algorithm achieves approximately 80%



(a) Classification percentages



(b) Training time for handwritten digits

Fig. 7. Comparison to k-means clustering.

classification rate. Using the k-means adjustment raises it noticeably to 87%. Plain k-means obtains the extremely good result of 95%. This is an expected result, since the cluster centers in k-means are not constrained by the search tree, but they can move more freely.

This freedom does come with a heavy cost, though, as we can see in subfigure (b). It lists the running times of ten cross-validated training/query rounds. Both versions of the Evolving Tree take only two minutes, while k-means take up to a half hour. This figure shows clearly the benefits of the search tree. The training time is only a fraction compared to regular k-means. All programs used in the timing tests were coded by ourselves in C++. This makes the running times comparable.

Another interesting thing to note is that the k-means adjustment to the Evolving Tree adds only a hardly noticeable increase to the total training time, yet it improves classification percentages noticeably. We can deduce that the k-means adjustment should be used with large databases. It can be hypothesized that we could obtain equivalent results without the adjustment. Unfortunately that would require a lot of trial and error with the parameters. The k-means adjustment offers a simpler way to achieve better results.

Overall, the Evolving Tree offers a classical trade-off to the k-means algorithm. Using it seems to lose some accuracy, but this is countered by the enormous decrease in computational complexity. This makes the Evolving Tree suitable for huge problems that have been unfeasible to approach with classical methods.

F. Visualization experiments

To obtain further insight into the behaviour of different algorithms, we did visualization experiments comparing ETree with regular SOM and k-means clustering. Since the Evolving Tree is designed for large dimensional data, we cannot plot them directly. We took the data and projected it to two dimensions using Sammon's mapping [20]. This preserves the overall shape of the data cloud. Since Sammon's mapping is nonlinear and in our cases reduces the dimensionality quite drastically, the resulting images should be analyzed quite critically.

Figure 8 (a) shows the locations of data vectors, ETree and SOM nodes, and k-means cluster centers. We have used the homogeneous texture data set which has 1300 vectors. Almost all of them are in the dark clump in the middle of the picture. Only roughly one to two hundred vectors are further away. Every algorithm has its own way of distributing nodes between these two areas.

First we look at the Evolving Tree, which focuses very strongly on the dense area. There are only a few nodes in the sparse data areas, but they follow the overall shape of the data quite well. The two furthest nodes cover only a few data vectors which are most likely outliers.

The SOM's rigid grid structure can be seen in the third picture as the nodes form regular structures. SOM focuses less on the dense areas than ETree. SOM's training formulas also prevent it from covering the entire data set, it is not as spread out as ETree. This is known as the *border effect* of SOM. Another consequence of the training is that there are a lot of nodes in the semi-sparse area up and right of the center. This area is likely overrepresented.

Last we have k-means which seems to have best captured the overall shape of the data cloud. There are, however, several outlier clusters which lie very far from the data cloud. These are not shown on the image due to scaling issues. Another potential drawback is that k-means seems to put more weight on the sparse areas than the data blob in the center. This behaviour can be justified in this case since k-means obtains the best classification results.

Figure 8 (b) shows the locations of ETree nodes per layer. They show how the tree grows outwards as the training progresses. The first two layers are rather plain. The next ones are more interesting. They show how the tree grows outward layer by layer. The large amount of splits at the dense central area can also be seen in every layer. Finally at layer six the tree only grows at the dense area. Weight decay prevents the

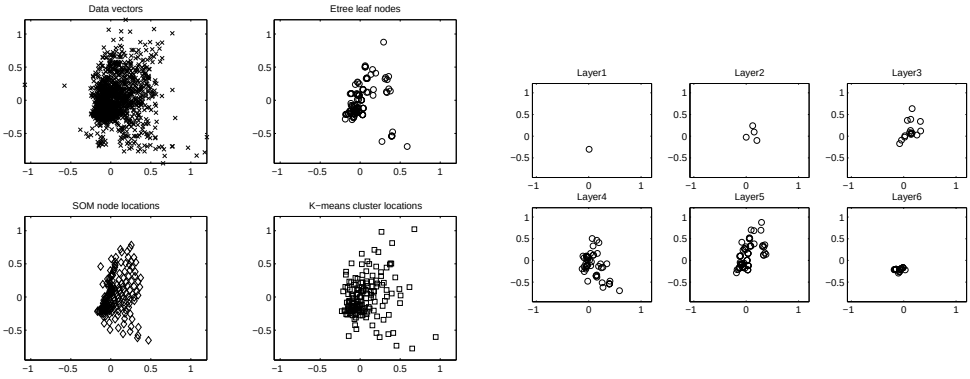


Fig. 8. Homogeneous texture data and node vector locations obtained with Sammon's mapping. Subfigure b shows the locations of ETree's nodes per layer.

outmost branches from growing. This behaviour is exactly as was predicted in the algorithm description.

Leaf node locations per class: Figure 9 shows the locations of Sammon projected data vectors for each of the 14 classes. There are no clear cluster differences and many of the classes overlap significantly. Given this very difficult data set, it is not surprising that none of the algorithms works properly for all cases. Class 12 demonstrates the very difficult nature of the data set. This particular run does not have a single vector in that class. This is unfortunate, but expected, since the defects in this class are almost identical to class 11 and there are only about 30 of them. ETree assigns nodes to these classes in some runs but not in others, resulting in the poor performance in Figure 5. The overall shape of each class is somewhat captured, but careful analysis locates several anomalies.

Take for example class 9, which is a small clump just below the middle area. All algorithms find this clump quite well, but both SOM and k-means have spurious nodes over the main cluster. Similarly class 5 is elongated to the top and right, but only SOM finds it. Overall the results mirror the above results for the total data set. The Evolving Tree has the tightest node locations, k-means is very spread out and SOM stands between these two.

G. Quality of clustering

One very common way of measuring different clustering algorithms is *quantization error*. It is usually defined as the average distance from a data vector to its nearest codebook vector. Given two similar methods, the one with the smaller quantization error is usually the better one. Figure 10 (a) shows the quantization error for the regular ETree, for ETree without the search tree, for SOM, and k-means clustering. The homogeneous texture was used for these experiments. K-means and SOM obtain noticeably smaller quantization errors, but the situation is not as straightforward.

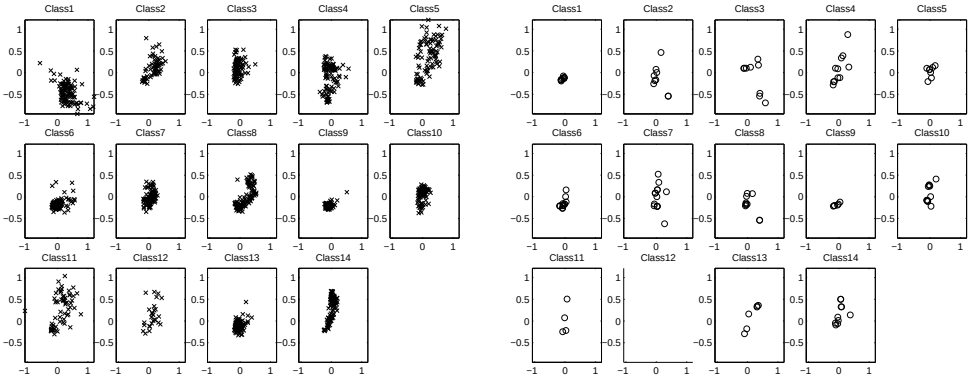
SOM and k-means are flat data structures, meaning that all nodes perform an identical task. The Evolving Tree, on the other hand, uses a hierarchical divide and conquer approach. The nodes at the top layers partition the data space with hyperplanes. In any non-trivial problem the clusters are non-separable so the partitioning will inevitably split some clusters non-optimally. This increases quantization error. This is apparent in the second column, removing the search tree halves the quantization error. This is the trade-off of hierarchicality: the ability to do fast operations causes some losses in accuracy.

This kind of error is inevitable in hierarchical data space partitioning. Therefore these kinds of algorithms can not obtain error levels of flat algorithms. The interesting question, then, is how much larger the error is. If the partitions are chosen badly, the error differences can reach an order of magnitude or more. The Evolving Tree does a tolerable job in this regard. The error is only about three times as large as with k-means. This is a satisfactory result, especially since the classification percentages are almost identical for both algorithms.

Quantization error measures the average distance from a data vector to a node. Adding nodes to an existing network can only decrease the error, no matter where in the data space they are placed. An efficient system has as few nodes as possible. To examine the efficiency we calculated the *inverse quantization error*, that is, the average distance from a node to its nearest data vector. If this value is much larger than normal quantization error, it implies the existance of outlier analysis nodes which may hinder computational efficiency. The results can be seen in the Figure 10 (b).

Etree and SOM have the smallest errors, while k-means' error is the largest. A large portion of k-means' error most likely comes from the outlier points mentioned in the previous chapter. In other data sets the inverse error was a lot smaller, some times even the smallest of the three. This shows that k-means is susceptible to outlier nodes. These don't usually matter, but they reduce the overall efficiency. Conversely ETree and SOM stay very close to the data cloud. It should be noted that the starved nodes in k-means can be detected and re-initialized. We have not done this because we wanted to test against the basic k-means algorithm and also because the reinitialization can be done in several different ways.

Finally we calculated the *Davies–Bouldin index* [21] for



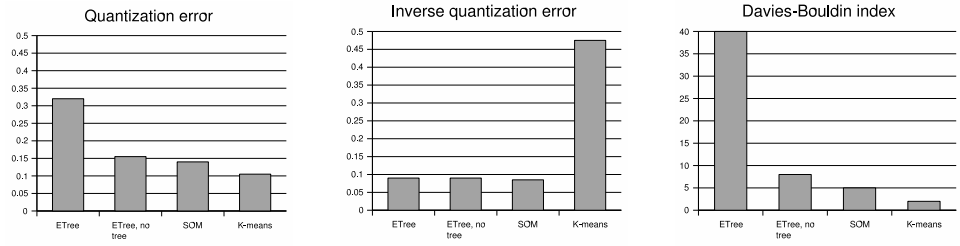
(a) Data vectors

(b) ETree leaf nodes

(c) SOM

(d) K-means

Fig. 9. Sammon projections of the data and the different algorithms.



(a) Quantization error

(b) Inverse quantization error

(c) Davies-Bouldin index

Fig. 10. Clustering results for homogeneous texture data set.

the different methods. It is one of the most common measures for examining clustering validity. Its basic assumption is that a good clustering consists of compact clusters that are well-separated. It is calculated using the following formula.

$$\mathrm{DBI} = \frac{1}{n}\sum_{i=1}^{n}\max_{i\neq j}\left(\frac{S_n(C_i)+S_n(C_j)}{S(C_i,C_j)}\right) \qquad (12)$$

Here $S_n(C_i)$ is the average distance of data vectors to the cluster center in cluster C_i. $S(C_i, C_j)$ is the distance between the centers of clusters C_i and C_j. A good clustering as described above should minimize this function.

The results are in Figure 10 (c). K-means obtains the best results. This is quite understandable, since the DB-index favors compact clusters that are well separated. This is what the k-means algorithm is trying to achieve as well. A much more interesting result is the search tree's effect on the Evolving

Tree. Using only the leaf nodes gives very roughly the same results as the SOM, but the entire tree's index value is a whole lot bigger. This can be caused by one of two things: either the clustering is bad or the clusters formed are not optimal in the DB sense, that is they are elongated and/or not well separated. Since the classification results are guite good, the clustering can not be totally bad. Therefore we can assume that the latter condition holds. This follows from the nature of the algorithm as well as the hierarchical structure.

H. Tree trunk reorganization

It has been established that the Evolving Tree does not always find the global BMU. One cause for this could be that the search tree is not optimal. To examine whether we could improve the trunk node locations after training we examined reshaping the tree's trunk. We developed a bottom-up algorithm that places every trunk node at the center of mass of its children. The reasoning is that this provides maximal separation between different branches.

In our tests this did not improve the classification rates at all, and on several occasions they decreased somewhat. We also calculated the mean quantization error and found that trunk reorganization increases it by an average of 5%–10%. These results are most likely caused by the hierarchichal nature of the algorithm. Upper layers partition the data and their children optimize their locations based on these partitions. When upper layer nodes are moved, the partition lines change. This causes changes in the data vectors mapped to the leaf nodes. Since the nodes are not in globally optimal places it is not very likely that the redistribution of data would lead to a better solution. Moving the trunk nodes would therefore require a much more sophisticated algorithm.

Above we have seen that even if we used the global BMU, the classification percentages remain mostly the same. The gains that could be obtained from tree reshaping would be quite minimal. These two experimental results suggest that the Evolving Tree's trunk does not require reshaping.

I. Comparison to other tree-shaped neural systems

So far we have compared ETree with classical non-hierarchical data analysis methods. It can be argued that these kinds of comparisons are not totally objective due to the different nature of the algorithms. We will now compare ETree with two other modern tree shaped neural systems, the S-Tree [22] and CNet [11]. Their main differences to ETree are that they don't use the neighborhood function and their splitting rules are more complex. CNeT also utilizes the class information in splitting the nodes which makes it a supervised algorithm.

All three systems are based on competitive learning and have tree structure that grows as the training progresses. Algorithmically S-Tree is the most complicated and ETree is the simplest. We examined both classification performance and training time. All the algorithms were coded in C++, so CPU time comparisons are fair. The parameters for all different systems were obtained by experimentation, the best result was chosen for every algorithm.

In Figure 11 we can see the classification percentages for the different algorithms. The results are consistent with earlier experiments. All three hierarchical systems outperform regular SOM. ETree and CNeT are very evenly matched, and their performance is very close to k-means (in Figure 7 (a)). S-Tree has the worst performance of the hierarchical methods, but the margin is quite small. On the larger digit database k-means enhanced ETree is clearly the best, CNeT holds second place with S-Tree very close to it.

Figure 12 shows the training times for the different algorithms. The times have been normalized by dividing them with ETree's training time. In all cases S-Tree is the slowest one. CNeT is slower on MPEG data but slightly faster than ETree on the digit database. Overall ETree and CNeT seem to have very similar time complexity.

The Evolving Tree has the best performance in almost all the tests which makes it the preferred algorithm of the three, especially in classification and clustering tasks.

V. Discussion

During our research we have discovered that the Evolving Tree is robust against disruptions in the algorithm. Its nature seems to automatically compensate the parameter variations. This can be a good thing or a bad thing. If the algorithm performs consistently well, there is little need to fiddle with the parameters. Whereas if the performance varies a lot, the ability to fine tune the performance is often essential. In our experiments the results have been relatively good, so this robustness has been a good thing.

One of the interesting paradigms noted in our experiments is power through simplicity. The Evolving Tree has been designed with a strict adherence to the KISS principle[1]. That is, every aspect of the algorithm has been kept as simple as possible, but no simpler. ETree is the simplest of the three compared hierarchical algorithms, yet it has the best performance. The most complex one (S-Tree) performs worst in these experiments.

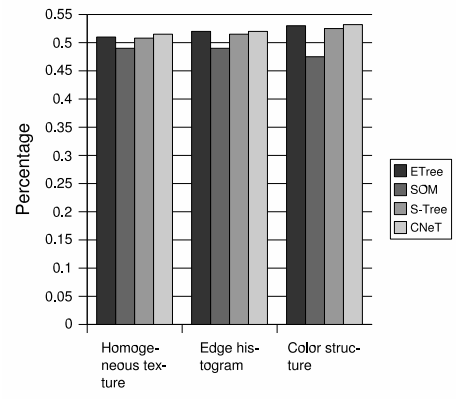
VI. Software package

We have released our implementation of the Evolving Tree as a free software package under the GNU General Public License [23]. We encourage other researchers to use and adapt the analysis package to their own problems.

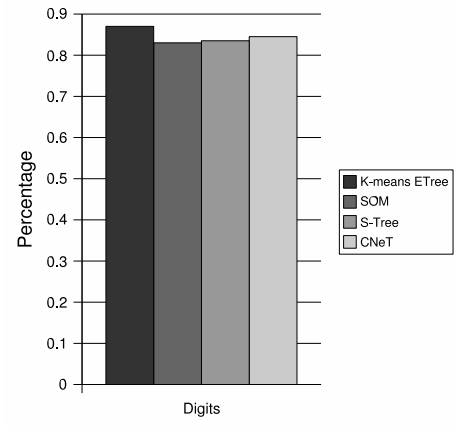
Our implementation is a very light weight C++ program which should run on almost any platform. It has been tested on Linux, IRIX and Tru-64. There are also several helper scripts coded in Python. These scripts make it easy to preprocess data, run tests and analyze the results. The package comes with extensive documentation. The documents include a user's guide, data format descriptions and so on. It also contains a hyperlinked reference documentation describing every class, function and file.

The package can be downloaded from `http://www.cis.hut.fi/research/etree/`.

[1] Also known as the "Keep it simple, scientist" principle.

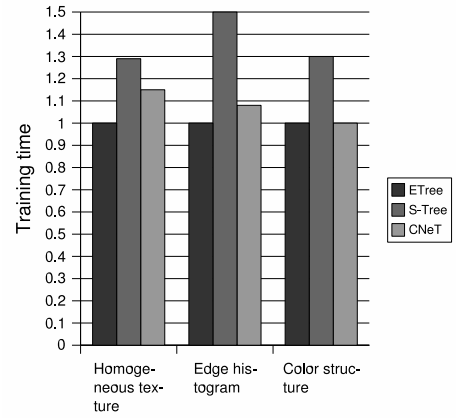


(a) MPEG-7 features

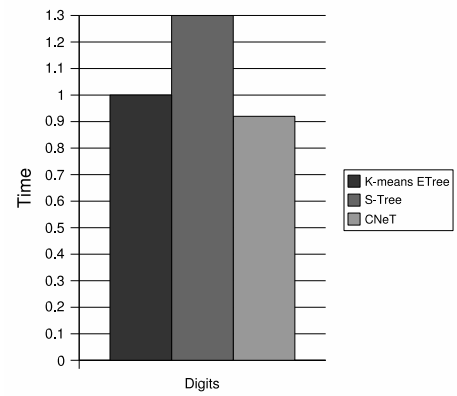


(b) Handwritten digits

Fig. 11. Classification percentages for ETree, SOM, S-Tree and CNeT.



(a) MPEG-7 features



(b) Handwritten digits

Fig. 12. Normalized training times for ETree, S-Tree and CNeT.

VII. Conclusions

We have analyzed and compared the Evolving Tree against many different systems. ETree's performance is quite close to classical, nonhierarchical algorithms but it is noticeably faster, in our tests an order of magnitude. Complexity analysis indicates that the difference becomes even larger as the size of the data set grows. This makes ETree suitable for several such tasks that have been too slow with nonhierarchical methods. We also find that ETree outperforms similar kinds of algorithms. It is also simpler, which is not only a virtue in its own right, but also makes implementation and application easier. Several improvements to the basic algorithm were suggested and analyzed. They, too, are relatively simple, but some of them still manage to improve the overall performance. The other changes illustrate that the basic algorithm is quite robust against alterations.

Acknowledgements

The authors would like to thank Mr J. Rauhamaa of ABB Oy for his help and comments. The financial support of the Technology Development Centre of Finland (TEKES's grant 40120/03) and ABB oy is gratefully acknowledged. We would also like to thank Petri Turkulainen for coding and running some of the experiments.

References

[1] T. Kohonen, *Self-Organizing Maps*, 3rd ed. Berlin: Springer, 2001.
[2] J. Blackmore and R. Miikkulainen, "Incremental grid growing: Encoding high-dimensional structure into a two-dimensional feature map," in *Proceedings of the IEEE International Conference on Neural Networks*, vol. 1, 1993, pp. 450–455.
[3] B. Fritzke, "Growing cell structures — a self-organizing network for unsupervised and supervised learning," *Neural Networks*, vol. 7, no. 9, pp. 1441–1460, 1994.

[4] J. Bruske and G. Sommer, "Dynamic cell structure learns perfectly topology preserving map," *Neural Computation*, vol. 7, no. 4, pp. 845–865, 1997.

[5] E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquín, "Searching in metric spaces," *ACM Computing Surveys*, vol. 33, no. 1, pp. 273–321, March 2001.

[6] P. Koikkalainen and E. Oja, "Self-organizing hierarchical feature maps," in *Proceedings of 1990 International Joint Conference on Neural Networks*, vol. II, San Diego, CA, 1990, pp. 279–284.

[7] V. J. Hodge and J. Austin, "Hierarchical growing cell structures: TreeGCS," *IEEE Transactions on Knowledge and Data Engineering*, vol. 13, no. 2, pp. 207–218, 2001.

[8] H.-H. Song and S.-W. Lee, "A self-organizing neural tree for large-set pattern classification," *IEEE Transactions on Neural Networks*, vol. 9, no. 3, pp. 369–379, May 1998.

[9] F. Luo, L. Khan, F. Bastani, I. Yen-Ling, and J. Zhou, "A dynamically growing self-organizing tree (DGSOT) for hierarchical clustering gene expression profiles," *Bioinformatics*, 2004. [Online]. Available: http://bioinformatics.oupjournals.org/cgi/content/abstract/bth292v1

[10] M. Dittenbach, A. Rauber, and D. Merkl, "Recent advances with the growing hierarchical self-organizing map," in *Proceedings of the 3rd Workshop on Self-Organizing Maps*, ser. Advances in Self-Organizing Maps. Lincoln, England: Springer, June 13-15 2001, pp. 140–145.

[11] S. Behnke and N. Karayiannis, "Competitive neural trees for pattern classification," *IEEE Transactions on Neural Networks*, vol. 9, no. 6, pp. 1352–1369, November 1998.

[12] J. Pakkanen, "The Evolving Tree, a new kind of self-organizing neural network," in *Proceedings of the workshop on Self-Organizing Maps '03*, Kitakyushu, Japan, Sept. 11–14 2003, pp. 311–316.

[13] J. Pakkanen and J. Iivarinen, "A novel self-organizing neural network for defect image classification," in *Proceedings of IJCNN 2004*, Budapest, Hungary, 2004, pp. 2553–2556.

[14] J. Pakkanen, J. Iivarinen, and E. Oja, "The Evolving Tree — a novel self-organizing network for data analysis," *Neural Processing Letters*, vol. 20, no. 3, pp. 199–211, December 2004.

[15] A. Guttman, "R-trees: a dynamic index structure for spatial searching," in *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM Press, 1984, pp. 47–57.

[16] C. Bishop, *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.

[17] B. S. Manjunath, P. Salembier, and T. Sikora, Eds., *Introduction to MPEG-7: Multimedia Content Description Interface*. John Wiley & Sons Ltd., 2002.

[18] J. Pakkanen, A. Ilvesmäki, and J. Iivarinen, "Defect image classification and retrieval with MPEG-7 descriptors," in *Proceedings of the 13th Scandinavian Conference on Image Analysis*, ser. LNCS 2749, J. Bigun and T. Gustavsson, Eds. Göteborg, Sweden: Springer-Verlag, June 29–July 2 2003, pp. 349–355.

[19] L. Holmström, P. Koistinen, J. Laaksonen, and E. Oja, "Comparison of neural and statistical classifiers — theory and practice," Rolf Nevanlinna Institute, Helsinki, Research Reports A13, 1996.

[20] J. W. Sammon, "A nonlinear mapping for data structure analysis," *IEEE Transactions on Computers*, vol. C-18, no. 5, pp. 401–409, May 1969.

[21] D. Davies and D. Bouldin, "A cluster separation measure," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 1, no. 4, pp. 224–227, 1979.

[22] M. Campos and G. Carpenter, "S-TREE: self-organizing trees for data clustering and online vector quantization," *Neural Networks*, vol. 14, no. 4–5, pp. 505–525, May 2001.

[23] Free Software Foundation, "The GNU general public license," http://www.gnu.org/licenses/gpl.html.

Jussi Pakkanen received his M.Sc. degree in engineering physics from Helsinki University of Technology in 2002. He is currently working as a researcher in the Lab. of Computer and Information Science, where he is furiously trying to finish his Ph.D. thesis. He is the coauthor of several articles, whose topics include image analysis, content-based retrieval, approximate indexing and unsupervised learning. His current research interests revolve around new data analysis methods that can be efficiently applied to huge data sets.

Jukka Iivarinen received his M.Sc., Lic.Sc.(Tech.) and D.Sc.(Tech.) degrees in computer science from Helsinki University of Technology in 1994, 1997, and in 1998, respectively. He is currently a researcher in the Lab. of Computer and Information Science, HUT. He is the vice chairman of the Pattern Recognition Society of Finland. He has acted as a referee in several conferences and journals, and is the coauthor of several conference papers and journal articles on image analysis, pattern recognition, and neural computing. His current research interests include neural networks and computer vision, especially applications of the self-organizing map in image segmentation, classification and retrieval.

Erkki Oja (S'75-M'78-SM'90-F'00) is the Director of the Adaptive Informatics Research Centre and Professor of Computer Science at the Laboratory of Computer and Information Science, Helsinki University of Technology, Finland. He received his Dr.Sc. degree in 1977. He has been research associate at Brown University, Providence, RI, and visiting professor at Tokyo Institute of Technology. Dr. Oja is the author or coauthor of more than 280 articles and book chapters on pattern recognition, computer vision, and neural computing, and three books: "Subspace Methods of Pattern Recognition" (RSP and J.Wiley, 1983), which has been translated into Chinese and Japanese, "Kohonen Maps" (Elsevier, 1999), and "Independent Component Analysis" (J. Wiley, 2001). His research interests are in the study of principal component and independent component analysis, self-organization, statistical pattern recognition, and applying artificial neural networks to computer vision and signal processing. Dr. Oja is member of the editorial boards of several journals and has been in the program committees of several recent conferences including ICANN, IJCNN, and ICONIP. He is member of the Finnish Academy of Sciences, Founding Fellow of the International Association of Pattern Recognition (IAPR), and past president of the European Neural Network Society (ENNS).