EnTS - a new Entropy-based Tree-indexing System

Sebastian Bader

supervisor: Dr. Frederic Maire

March 2002

"They are very treelike -
If you do not look carefully, you would think
that they are trees." (Gandalf about ents, in [Tol66])

s.bader@qut.edu.au

This paper describes a new indexing-system, which can be used to enhance the speed of almost every algorithm working on codebooks. The only requirement to use EnTS is the existence of a metric vector-space containing the data-items. But as most algorithm work on n-dimensional euclidean vectors, this is not a big demand.
During the development, the work was focused on vector-quantisation. But the system is capable to speed-up other groups of algorithms as well.
EnTS helps to increase the speed of other algorithms without loosing lot of the quality. The System outperforms a number of similar approaches.

supervisor: f.maire@qut.edu.au March 2002

Contents

List of Algorithms

1 Motivation

With the development of new techniques for data collection the amount of data to be processed increased rapidly. But processing of these data is necessary, e.g. for knowledge-discovery, pattern recognition, function approximation or vector quantization. Unfortunately, most algorithms developed to process the data do not scale very well into larger problems and therefore they are not applicable. They work very well for small data-sets, but the databases of the Human Genome Project or the S.E.T.I-project contain terra-bytes of data, i.e. billions of high-dimensional vectors.

A lot of the algorithms which work on large datasets need to create a codebook, which contains a set of vectors (prototypes or reference-vectors). These codebooks are generated and modified to extract the knowledge from the input-data and to abstract from the data itself. A well known problem is to find the closest codebook-vector (winner) for a given input. For instance, this is necessary for most vector-quantization algorithms (LBG, LBG-U, GNG, ...). A quantizer is an algorithm/machine which computes a set of reference-vectors to minimize a given error-function. The error-function is defined as a function of the distances between each input-vector and its closest reference-vector.

But as most implementations store the codebooks in arrays a tabular search is necessary to find the winner. Therefore the time-complexity is linear in the size of the codebook.

For ordered sets (e.g. natural numbers, ...) it is possible to create an indexing-structure that provides logarithmic access into the data. Unfortunately, there does not exist a total order, which is compatible with the algebraic vector operations, for the data-spaces used in the applications mentioned above. Therefore standard search trees are not applicable and more sophisticated methods become necessary. Furthermore, most quantizer algorithms require an update of the codebooks. I.e. even though existing static indexing-systems are useful to provide fast access into the processed (static) data, these structures are not suitable for adaptive codebooks.

This paper proposes a new indexing-system that is suitable for any quantizer working on a metric vector-space. It provides logarithmic access into sets without a natural order and can be used to improve existing algorithms without changing the algorithms themselves.

This paper is organized the following way: The next section will give an introduction into indexing-systems, vector-quantization and some of the algorithms used to develop and test EnTS. Section 3 will describe the developed system. Afterwards the results are presented and the system is compared to existing aproaches. The paper finishes with the conclusion and a discussion of the future work.

2 Introduction & Background

2.1 Indexing Systems

As mentioned above, the problem is to provide fast access to the reference-vectors (also called codebook-vectors) stored within the codebook. A fast indexing-system would improve the speed of many machine learning algorithms.

But an indexing-system should not only provide fast access to the vectors stored. It should be independent from the algorithm it is used in, i.e. it should work like a plug-in for every algorithm working with codebooks. Furthermore, it should not influence the results of the client algorithm - only make it faster.

An indexing-system can be seen as a stand-alone tool. It must be able to retrieve the closest (and second-closest) item stored and to modify the codebook. There are three mutators for the modification: *insert*, *delete* and *update*. The first one modifies the codebook by inserting a new reference-vector into it, while the delete-operation removes a given reference-vector. The update operation changes a reference-vector by assigning a new position to it. And most important there must be a *query*-operation. This one returns for a given input-vector the closest and second-closest reference-vector stored in the codebook. These definitions enable us to define the abstract data-type shown in Figure 1.

Indexing-system: an abstract data-type

Constructors:
- *Empty:* $\rightarrow$ codebook
- *New:* set-of-vectors $\rightarrow$ codebook

Accessors:
- *Query:* codebook $\times$ input-vector $\rightarrow$ {first-closest, second-closest}

Mutators:
- *Insert:* codebook $\times$ new reference-vector $\rightarrow$ codebook
- *Delete:* codebook $\times$ reference-vector $\rightarrow$ codebook
- *Update:* codebook $\times$ reference-vector $\times$ new position $\rightarrow$ codebook

Figure 1: Indexing-system: an abstract data-type

2.2 Tree-indexing

Tree-structures are a common tool to implement indexing-systems (e.g. R-Trees [Gut84], R^+-Trees [SRF88], K-Trees [Gev], HiGS [VB96]). They can provide access with a running-time which is logarithmic in the size of the codebook, by splitting the complete search-space into smaller, (nested) cells. I.e. the tree-structure guides the search at each level into the half-space (or cell) containing the input-vector.

To provide a minimal expected access-time the tree should be as balanced as possible. I.e. the subtrees of any node should have about the same height.

2.3 Vector-quantization

Given a probability distribution P on a vector-space and an integer k, vector-quantization tries to find k codebook-vectors such that the quantization-error (distortion):

$$E = \int \|x - c(x)\|^2 p(x) dx$$

is minimal. The vector $c(x)$ is the closest codebook-vector for x and $p(x)$ the probability of x. I.e. by modifying the positions of the codebook-vectors the algorithm tries to minimize the sum of the distances of every input-vector to its closest codebook-vector.

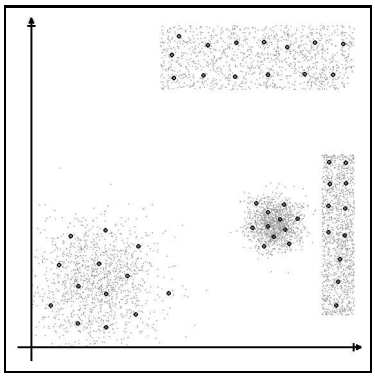


Figure 2: A codebook (large dots) for the given input-distribution (displayed as a sample - small spots)

Unfortunately, there is no way known to compute these codebooks directly from the input-distribution. But a lot of algorithms were developed to find good solutions. A few of these algorithms will be described in the following sections.

The search for the optimal solution can be seen as a search for the global minimum of the distortion-function. To find this minimum most of the algorithms use local optimizations. I.e. they use information derived from their current state to find the minimum. Unfortunately, they usually run into local minima before finding the global one.

The quantization-error can be used to compare different algorithms or different runs of the same algorithm. To evaluate an algorithm, the average quantization-error and its variance are compared with results of other algorithms.

2.3.1 LBG, GLA & k-means

LBG stands for Linde, Buzo and Gray - the authors of the algorithm, GLA for Generalized Lloyd Algorithm and k-means points out the idea of determining k means for a given set of vectors. All these names refer to one algorithm. It is one of the classical approaches for vector-quantization and has influenced a lot of other solutions.

This algorithm works on finite sets of input-vectors and updates the position of every codebook vector depending on its local environment, i.e. the

input-vectors of its neighborhood. A finite set is necessary to enumerate the neighborhood.

The algorithm starts with distributing k codebook-vectors according to the input-distribution. These codebook-vectors are repeatedly moved into the centers of their Voronoi-regions until they do not change their positions any more. The Voronoi-region for a given vector c_i from a set of vectors $C = \{c_1, \dots c_n\}$ is defined as the set of vectors (from the input-space) for which c_i is the closest one with respect to C. I.e. every codebook-vector is moved to the center of the input-vectors it is responsible for. Each of these steps is called Lloyd-Iteration (see Figure A1).

Lloyd-Iteration (*Input-Set I,Codebook C*)
returns *Codebook*

1. Create $z = |C|$ empty sets (Voronoi-sets V_c) - one assigned to each codebook-vector $(c_1 \dots c_z)$.
2. Determine for every input-vector x from I the closest vector $c(x)$ from C and add x to the Voronoi-set $V_{c(x)}$.
3. Compute the centroid for every Voronoi-set and assign it to the corresponding codebook-vector $c_i = \sum \{x | x \in V_{c_i}\} * \frac{1}{|V_{c_i}|}$
4. Return the updated codebook as solution.

Figure A1: Lloyd-Iteration

When the positions of the codebook-vectors change, their Voronoi-regions (and therefore their Voronoi-sets) change as well. Therefore it is necessary to repeat the process until the positions of the codebook-vectors are stable (see Figure A2).

LBG (*Input-Set I, Size of the Codebook k*)
returns *Codebook*

1. Create a codebook C containing k vectors and initialize them to random positions according to I.
2. Repeat until C does not change:
   $C = Lloyd\text{-}Iteration(I, C)$
3. Return the codebook.

Figure A2: LBG

Starting from a fixed set of input-vectors, there will always be a state in which the algorithm stops, because each step decreases the quantization-error and its value is bound by 0.

The number of iterations necessary depends a lot on the number of input-signals and the size of the codebook. Unfortunately, the results differ a lot, depending on the initialization of the codebook-vectors. I.e. multiple executions of the algorithm may lead to very different results.

2.3.2 LBG-U

As mentioned above, the results of LBG differ a lot depending on the initialization. This is due to the fact that the algorithm runs into local minima because the reference-vectors are adapted according to their local environments only. To overcome this drawback, Fritzke introduced the idea of utility in combination with non-local jumps. If a certain reference-vector is currently not very useful, it is moved to a position where it might help more. These non-local jumps enlarge the search-space of the algorithm.

The problem is to find a measure of utility. If two reference-vectors have nearly the same position, this arrangement is not very useful because it is possible to remove one of them without a large increasing of the error. Therefore, one of them might be more useful at another position. The same holds for reference-vectors which are the winner for only a few input-vectors. These reference-vectors neither cause a large local error and can be used more efficiently as well.

On the other hand, a vector with a large Voronoi-region or with a lot of input-vectors creates a big error and needs help. These vectors are candidates for jump-targets. I.e. vectors with a very close neighbor or very small Voronoi-regions are moved close to vectors which are responsible for a big region or a lot of vectors.

To formalize these two concepts ('not useful' and 'need help') two values are attached to every codebook-vector. The first contains the local quantization-error, i.e. the sum of the squared distances to all the vectors it was winner for:

$$E_i = \sum_{x_j \in V_{c_i}} \|c_i - x_j\|^2$$

The other value is used for the utility. It accumulates the difference of the quantization error when the reference-vector would be removed. If a reference-vector is removed the second-closest becomes the winner for the input-vector, i.e. the error-value caused by this input-vector would be a bigger one. This increase of error can be interpreted as utility of the reference-vector.

$$U_i = \sum_{x_j \in V_{c_i}} \|c_2(x_j) - x_j\|^2 - \|c_i - x_j\|^2$$

$c_2(x)$ denotes the second-closest reference-vector for the input-vector x.

Both values are accumulated over one epoch (processing of every input-signal) and set to 0 before. LBG-U performs a complete LBG and afterwards the reference-vector with the smallest utility is moved close to the vector with the highest error. This process (LBG + jump) is repeated until the error of the LBG-process will not improve any more. Sometimes a jump might increase the error. In these cases a copy of the codebook that was stored before the last jump is returned as the result (see Figure A3).

As mentioned above, the non-local jumps enlarge the search-space of the algorithm and allow to escape from local minima where LBG gets stuck. Therefore, they lead to better results with a smaller variance.

As each LBG-U performs at least one LBG, the results can't be worse and every following iteration creates a better result. The price for better results is a longer computation process, because multiple LBG's might be performed.

LBG-U (*Input-Set I, Size of the Codebook k*)
returns *Codebook*

1. Create a codebook C containing k vectors and initialize them to random positions according to I, initialize the quantization error $E = \infty$.
2. Create a copy of the current codebook $K = C$ and the current quantization error $F = E$.
3. Repeat until C does not change:
   $C = \mathit{Lloyd\text{-}Iteration}(I, C)$, i.e. perform an LBG-step
   and accumulate the local error- and utility-values $(E_i, U_i, E = \sum E_i)$.
4. If $E \geq F$ return the stored copy K.
5. Copy the current codebook $K = C$ and the quantization error $F = E$.
6. Determine the least useful reference-vector $(u = \arg\min_{c \in C} U_c)$ and the one with the highest error-value $(e = \arg\max_{c \in C} E_c)$ and move u close to e.
7. Continue with 3

Figure A3: LBG-U

2.3.3 GNG

For most of the quantization-problems the best fitting codebook-size is not known in advance. To solve this problems with LBG(-U) it would be necessary to guess a certain size, execute the whole algorithm and try it again until the results are good enough. To overcome this drawback, some algorithms were developed that work on a codebook with changing size.

Growing Neural Gas (GNG) is one of them. It is an extension of the Neural Gas by Martinetz and Schulten.[MS91] The extension was done by Fritzke [Fri91]. Furthermore, it does neither require a finite set of input-signals (like LBG, LBG-U).

Starting with a set of two reference-vectors, the algorithm works similar to Kohonens Self-Organizing-Map [Koh90]. For each input-vector the closest codebook-vector is determined and moved a little towards the input (according to a given adaptation-rate ϵ_b). Furthermore, the second-closest is adapted as well, but with a smaller rate ϵ_n. Between these two codebook-vectors a connection (edge) is created. These edges are used to control the insertion of new reference-vectors and can be interpreted to get a better understanding of the underlying input-distribution. They also create a topological structure on the codebook, that can help to identify clusters within the input. This idea was introduced as Competitive Hebbian Learning in [MS91]. An example can be seen in Figure 3.

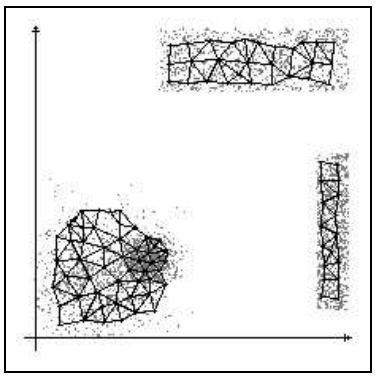


Figure 3: A codebook with edges for the given input-distribution (displayed as a sample - small spots)

Furthermore, each edge has an assigned age. This value is increased by one for every edge starting from the closest reference-vector. And if the age of one of the edges exceeds a certain value, the edge is removed. If the removing leads to reference-vectors without any connection to others, these vectors are removed as well. The whole algorithm is shown in Figure A4.

GNG (*Input-Distribution I, Maximum size of the Codebook k, Adaptation rates ϵ_b and ϵ_n, Maximum age a_{max}, Frequency of updates λ, Error-adaptation rates α and β, Stop-criterion*)
returns *Codebook*

1. Initialize the codebook C with two reference-vectors.
2. Generate an input-vector ξ according to I.
3. Determine the closest (c_1) and second-closest (c_2) reference-vector.
4. Create a connection between c_1 and c_2. If there already exist a connection, set its age to 0.
5. Add the squared distance between ξ and c_1 to the local quantization-error of c_1.
$$E_{c_1} = E_{c_1} + \|\xi - c_1\|^2$$
6. Adapt c_1 and c_2 according to ϵ_b and ϵ_n.
$$c_1 = c_1 + \epsilon_b(\xi - c_1)$$
$$c_2 = c_2 + \epsilon_n(\xi - c_2)$$
7. Increase the age of all edges starting in c_1 by one.
8. Remove every edge with an age greater then a_{max}. If it leads to codebook-vectors without any connections, remove them as well.
9. If the number of processed input-vectors is a multiple of λ, insert a new codebook-vector:
   (a) Determine the codebook-vector with the greatest error-value $q := \arg\max_{c \in C} E_c$ and among its topological neighbors the one with the greatest error $f := \arg\max_{c \in Neighbors} E_c$.
   (b) Add a new reference-vector r into the codebook. Its position is the middle of the segment (q, f).
   (c) Remove the connection between q and f and add new ones for the pairs (q, r) and (r, f).
   (d) Update the local quantization errors
$$E_q = E_q - \alpha * E_q$$
$$E_f = E_f - \alpha * E_f$$
$$E_r = (E_q + E_f)/2$$
   (e) Reduce the quantization error of every codebook-vector to
$$E_c = E_c - \beta * E_c$$
10. If the stop-criterion (size of codebook, quantization-error, ...) is fulfilled return the current codebook, else continue with step 2.

Figure A4: GNG

2.3.4 GNG-U

This section will introduce an extension of GNG. Applying the concept of utility, GNG-U can be derived from GNG.

To each of the reference-vectors a utility-value is assigned. This value is computed/accumulated as described in section 2.3.2 as the difference between the squared distance to the winner and to the second-closest codebook-vector. I.e. for each input-vector the closest and second-closest codebook-vector are determined. For each of them the distance to the input-vector is computed. The utility of the winner is increased by the difference of these distances.

If the fraction of Error and Utility of a certain codebook-vector is smaller than a user-defined constant r, the codebook-vector is removed. If this leads to codebook-vectors without any connections, these are removed as well.

The modifications are applied to the GNG algorithm as described in Algorithm A4 as follows:

- Step 5 is extended to accumulate the utility as well.
- A "delete"-step is inserted after step 9.

The modified algorithm is shown as A5.

GNG-U (*Input-Distribution I, Maximum size of the Codebook k, Adaptation rates ϵ_b and ϵ_n, Maximum age a_{max}, Frequency of updates λ, Error-adaptation rates α and β, Stop-criterion, Relation of Error/Utility r*)
returns *Codebook*

⋮ (1..4 like in GNG)

5 Add the squared distance between ξ and x_1 to a local quantization-error of x_1. And update the utility.

$$E_{x_1} = E_{x_1} + \|\xi - x_1\|^2$$
$$U_{x_1} = U_{x_1} + (\|\xi - x_2\|^2 - \|\xi - x_1\|^2)$$

⋮ (6..9 like in GNG)

10 If the quotient of Error and Utility of a certain reference-vector is smaller than r ($E_c/U_c < r$), this node (c) is removed. Reference-vectors without any connection are removed as well.

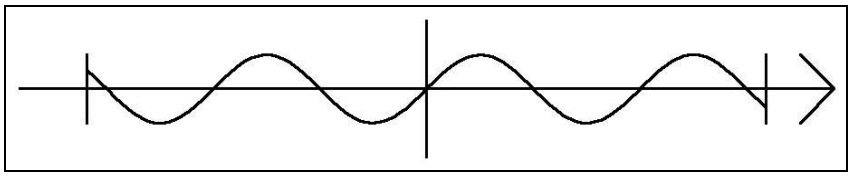
11 If the stop-criterion (size of codebook, quantization-error, ...) is fulfilled, return the current codebook, else continue with step 2.

Figure A5: GNG-U

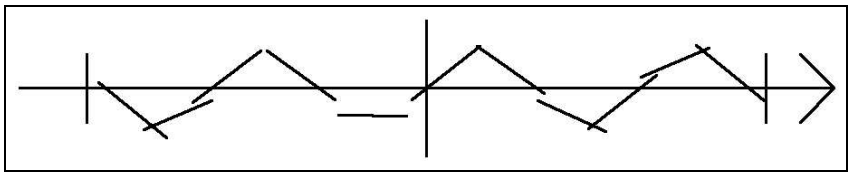
2.4 Local maps

Given a set of pairs $\{(x_i, y_i) \in (X, Y)\}$, containing an input-vector and an output-value, local maps try to approximate the underlying function with a set of local functions $\{f_i : X \to Y\}$. I.e. to each of these functions a position-vector p_i is assigned, which represents the location of the function. The result of the algorithm is the set of local functions, which can be used for approximations $A(x)$, computed by combining the weighted local functions $A(x) = \sum w_i * f_i(x)$. The weights depend on the distance of the input-signal and the functions' positions and can be seen as fuzzy membership functions. I.e. the closer a functions position vector is to the input, the more important is its result. Sometimes it is sufficient to take only the output of the closest function into account or an interpolation of the n closest. Figure 4 shows an example of such an approximation.

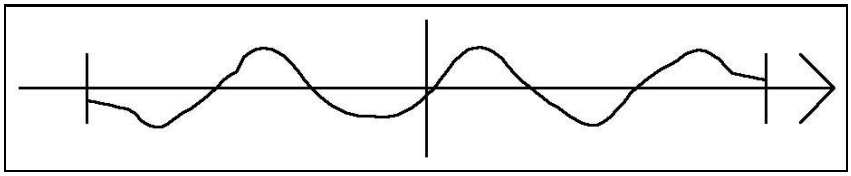
The class of function is predefined (e.g. linear functions), but all free parameters incl. the position of these functions are determined by the system.



(a) target-function



(b) local-map



(c) approximation

Figure 4: Local-Map: the target-function was used to create the input-pairs, b) shows the computed local functions and c) shows an approximation of the target-function computed by interpolating between the values of the two closest functions

Most algorithms update one or two functions only in each step. I.e. first the algorithm determines the functions which are responsible for a certain input-signal and afterwards these functions are updated. The vectors representing the positions can be seen as a codebook and therefore accessed via EnTS to speed up the process.

3 EnTS

This section describes the structure used to construct the indexing-system. It shows how the construction and modification of the internal structure works and how the system is meant to be used. Furthermore, it describes problems of the system and the corresponding solutions.

The systems interface As mentioned in section 2.1, an indexing-system can be described as an abstract data-type. This datatype defines the interface to the system (*Query*, *Insert*, *Delete* and *Update*). Therefore, it can be seen as a black box containing a set of vectors (the codebook) and providing access to them.

System implementation To implement a fast indexing-mechanism, a binary tree-structure is used. This decision-tree splits the input-space recursively into half-spaces. Therefore, each path from the root of the tree to a leaf-node contains the description of a polyhedron. The combination of all these polyhedras creates a partition on the input-space as shown in Figure 5.

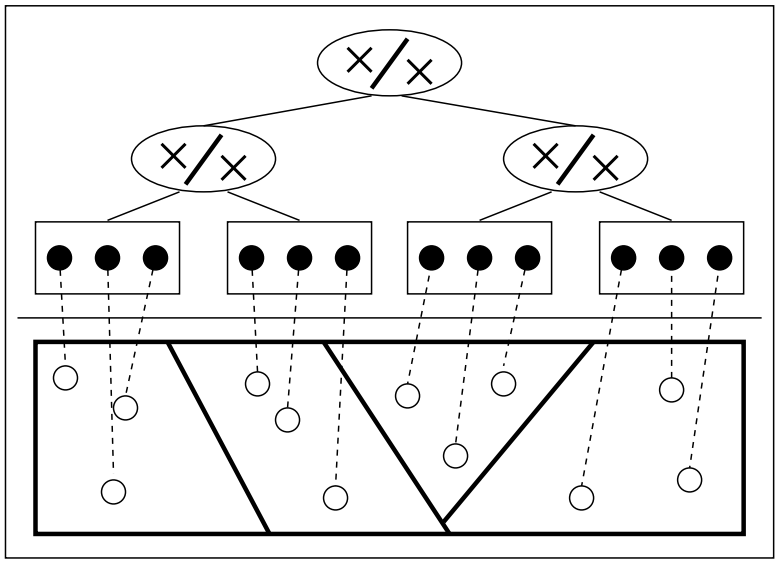


Figure 5: The tree-structure used in EnTS including the partition created

The leaf-nodes of the tree contain the stored vectors. Each of these nodes hold a certain amount of vectors, which is spezified by a user-parameter. It is possible to set this parameter to 1, i.e. each leaf-node can contain maximally one reference-vector. But it is more useful to set it to a higher value like 10, because for small sets of vectors a tabular search is faster, than a search directed by a tree-structure. Furthermore, it is easier to balance the tree if there is some tolerance in the number of vectors stored in the leaf-nodes. (For a further discussion see section 3.3.)

Each internal node holds a decision-rule which decides in which subtree to step while searching for a certain vector. These decisions are constructed of

$$Hyperplane: h = \{x | x * n = d\}$$

$$left - side: \; l = \{x | x * n > d\}$$

$$right - side: r = \{x | x * n \leq d\}$$

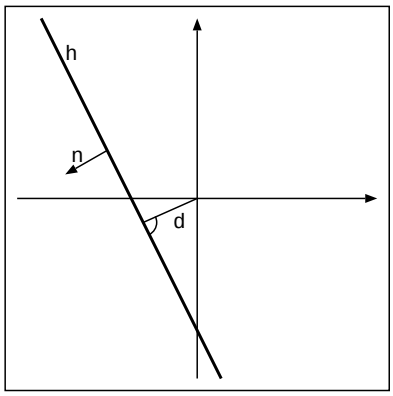


Figure 6: Picture of a hyperplane and the decision

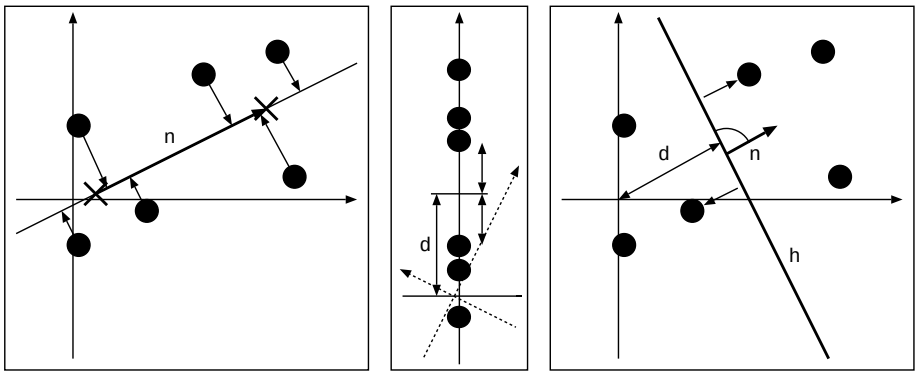
hyperplanes (linear discriminant functions) splitting the space into half-spaces. Each hyperplane is described by its normal-vector and the distance to the origin of the space (Figure 6). To each subtree one side of the hyperplane is assigned. If the input-vector is on the side containing the origin, the search will continue in the right subtree, in the left one otherwise.

3.1 Generation of the Indexing-System

Starting from a set of vectors to be indexed, the algorithm performs a recursive split. These splits are done by a hyperplane as described above. To find a good-splitting hyperplane the principal component of the set of vectors is used to determine the direction. Once this direction is found, it is easy to find the best-splitting distance to the origin. The calculation of the main component could be done by a PCA. But in EnTS a 2-means is performed and the vector connecting the two means is used as the main-component, because it is computational more efficient than a PCA.

Once the direction of the hyperplanes normal vector is found, the distance d to the origin needs to be calculated. Therefore, all vectors are projected onto the normal-vector using the inner-product. These products are sorted and the median (or the mean of the two medians) is used for the offset of the hyperplane. The median is used to create a 50/50-partition. This leads to a tree as flat as possible, which guarantees an optimal average access time. Both subsets created by this partition are processed recursively until each set contains a maximum of a user-defined number of vectors.

Another way to construct the indexing-system is to build it incrementally. By starting with one empty leaf-node the reference-vectors are inserted into the system as described in section 3.2 in the *Insert*-part.



(a) the two means (the $\times$) and calculation of the normal-vectors direction n

(b) projection of all vectors onto the normal-vector and calculation of the distance d to the origin

(c) the computed hyperplane ($h = \{x | x * n = d\}$) incl. the normalized normal-vector ($|n| = 1$)

Figure 7: Construction of a hyperplane

3.2 Operations of the Indexing-System

This section will describe in detail how the operations defined in section 2.1 are implemented within the EnTS. The operations required for an indexing-system are: *Query*, *Insert*, *Delete* and *Update*.

***Query* (codebook $\times$ input-vector $\rightarrow$ {first-closest, second-closest}):** The Query-operation is used to find the nearest neighbor for a certain input-vector (and the second-closest). This search is executed in two steps. As the tree creates a polyhedral partition, the first step is to identify the polyhedron which contains the input-vector. Each leaf-node can be viewed as a polyhedra, that is defined by the hyperplanes contained in the nodes from the root to the leaf-node. Once the correct leaf-node polyhedron is identified a tabular search is performed to find the closest (and second closest) reference-vector.

But sometimes the search fails to return the closest codebook-vector, as the search is restricted to the cell containing the reference-vector. As shown in Figure 8, the closest reference-vector might be contained in a different leaf-node. To solve this misclassification-problem, the algorithm steps into two subtrees in doubt and performs a parallel search. But this process is described in more detail in section 3.3.

***Insert* (codebook $\times$ new reference-vector $\rightarrow$ codebook):** To insert a new reference-vector into the current codebook, EnTS first finds the correct leaf-node and inserts the vector into it. The search for the correct leaf-node is described above, while talking about the *Query*-Operation. As each leaf-node holds a set of vectors the new reference-vector is simply inserted into it.

But as mentioned before, each of these nodes must not hold more than a certain amount of vectors. I.e. the insertion may lead to an overflow of the leaf-node. To solve this problem, the EnT-System splits that node as described in section 3.1. I.e. a 2-means is performed giving the direction of a new splitting

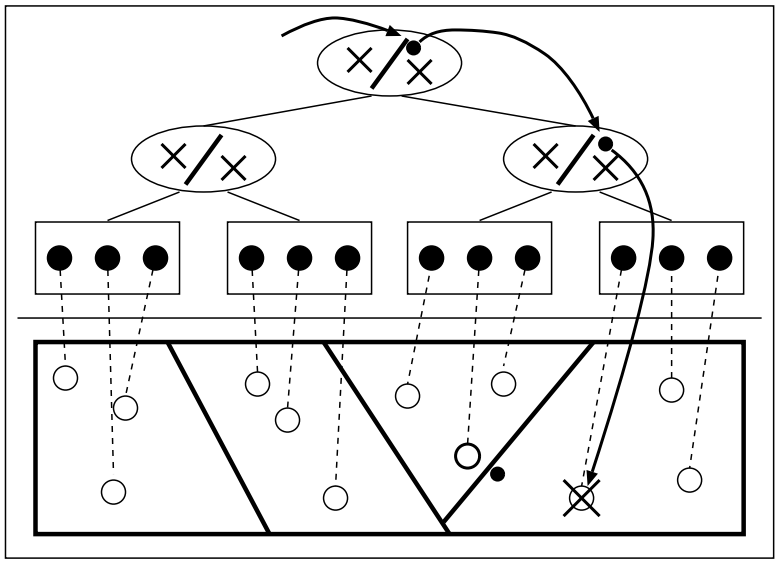


Figure 8: Misleading of the tree-structure. The closest reference-vector is marked by a thick circle and the wrong solution by the cross.

hyperplane and afterwards the whole set is split.

The old leaf-node is replaced with a new decision-node and the two corresponding leaf-nodes (see Figure 9). Unfortunately, this process may lead to an unbalanced tree, but the problem of rebalancing will be discussed in detail in section 3.4.

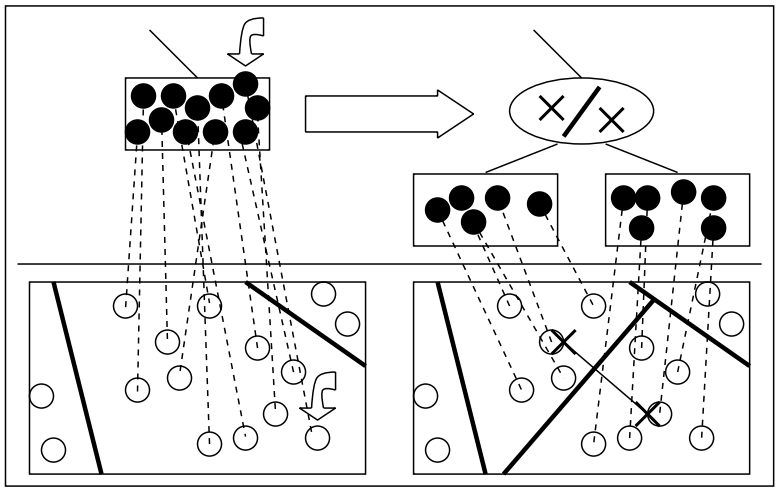


Figure 9: A split of a leaf-node caused by an overflow.

***Delete* (codebook × reference-vector → codebook):** As for insertion the first step of a deletion is to find the correct leaf-node, i.e. the one that contains the vector to delete. Once that node is found, the vector is simply removed from it.

If this vector was the last in its leaf-node, that node will be removed completely. Furthermore, its parent might get removed as well, as its decision is not useful anymore. I.e. the parent-node is replaced with the leaf-nodes sister, as shown in Figure 10.

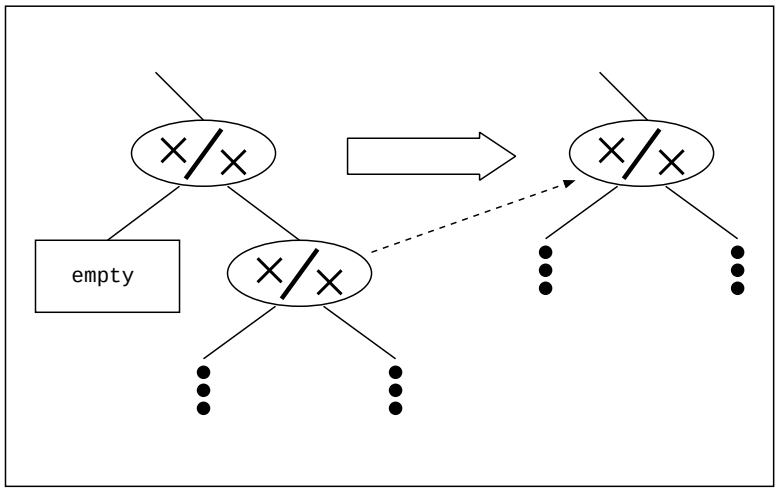


Figure 10: Deletion of an empty node and the replacement of its parent with the sibling.

To keep the tree as shallow as possible, it is useful to merge subtrees if they contain too few reference-vectors. If the deletion in a certain subtree leads to a total number of vectors that is smaller than the number a single leaf-node can hold, this subtree is replaced with the single node, as demonstrated in Figure 11.

Unfortunately, the delete-process may lead to an unbalanced tree. As mentioned above, the problem of (re-)establishing the balance will be discussed in detail in section 3.4.

***Update* (codebook × reference-vector × new position → codebook):** While updating the position of a reference-vector, the tree must be updated sometimes as well. First the system finds the leaf-node containing the old vector and the target-leaf-node. If these two nodes are identical, only the position of the vector is changed.

Otherwise the system performs a deletion followed by an insertion. I.e. in that case the reference-vector is removed from the system, its position modified and afterwards reinserted.

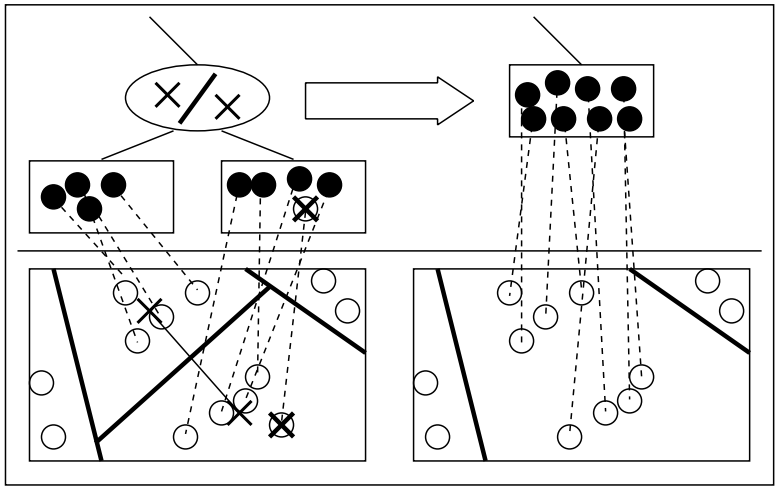


Figure 11: Deletion of a reference-vector and the merging of a subtree, which contains to few elements.

3.3 The partition and the No-Man's-Land

Given a set of reference-vectors $V = \{v_1, ..., v_n\}$ distributed in a space S it is possible to label each point in S with the number of its closest reference-vector. This process is called Voronoi-Tessellation.

The partition created by that tessellation would be the optimal solution for our problem, i.e. having a description of the borders of it, it would be easy to construct an indexing-system that works optimally. But the tessellation is neither computable in an acceptable time nor representable in a nice form (at least not for high-dimensional spaces). Unfortunately, the partition created by EnTS is not equal to the Voronoi-partition, as shown in Figure 12. Some of the Voronoi-regions are split by the EnTS-decision-borders. For input-vectors contained in the 'cut of' areas of a Voronoi-region (marked in Figure 12) EnTS returns the wrong reference-vector.

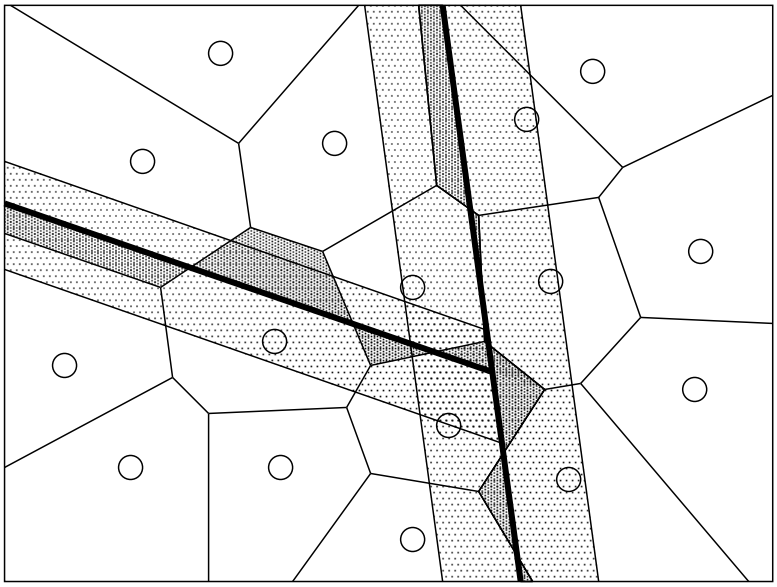
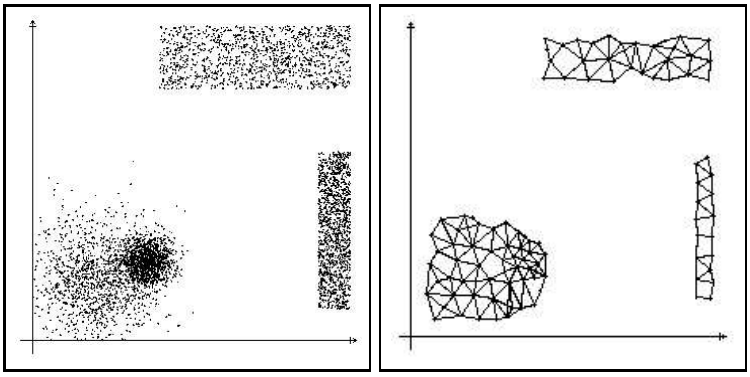


Figure 12: A set of reference-vectors (circles), their Voronoi-Tessellation (thin lines) and an EnTS-Partition (thick lines). The areas for which EnTS would return a wrong 'closest' reference-vector are marked dark-gray ('cut-off'-areas) and the 'No-Mans-Land' light-gray.

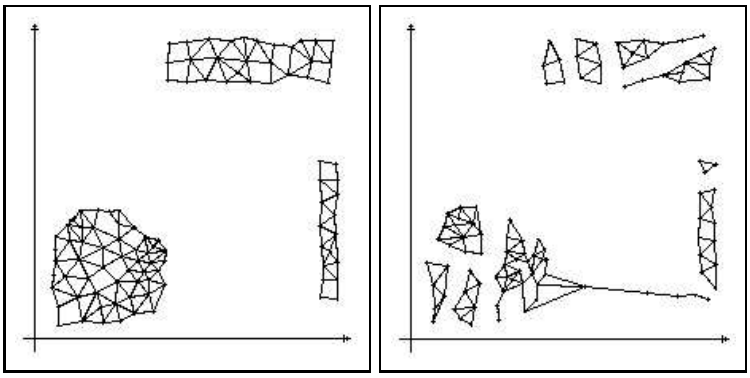
To solve this problem, we introduced the concept of the 'No-Man's-Land'. This can be understood as a fuzzification of the decisions. If a certain input-vector lays 'too close' to a border, the search is continued on both sides. To have a measurement of closeness, the system keeps track of the current mean for each subtree. These means and their distance to the decision-hyper-plane are used to decide whether to step into both subtrees or not. A certain fraction of that distance is considered to be the mentioned 'No-Mans-Land' (see Figure 12). This is a user-defined parameter and it controls the trade-off between accuracy and speed of the system. If the 'No-Man's-Land' has a width of 0, the system would be very very fast, but would not find the correct nearest neighbor for some input-vectors. While using a 'No-Man's-Land'-fraction of ∞, the system

would act like an array-implementation and always return the (correct) closest reference-vector. But, unfortunately, it would not be faster anymore, because every leaf-node is searched. Figure 13 shows different results of GNG.



(a) Vectors generated by the underlying input-distribution.

(b) The 'optimal' solution, computed with an array-implementation of the codebook (t=90sec).



(c) EnTS-solution with a No-Mans-Land-fraction of 0.3 (t=80sec).

(d) EnTS-solution with a No-Mans-Land-fraction of 0.0 (t=30sec).

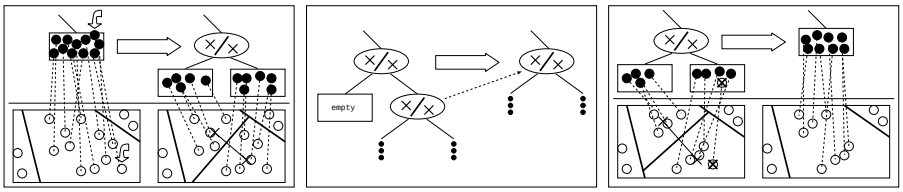
Figure 13: Results of GNG with different parameters.

But section 4 will give a more detailed discussion of the parameters. As described there, the concept of the 'No-Mans-Land' is a good and effective way to solve the misleading-problem created by the non-Voronoi-partition of EnTS.

3.4 Balance of EnTS

As mentioned above, it is important to keep the tree as small and balanced as possible to provide the optimal average access-time. This section first gives a definition of (un-)balance and afterwards describes how the balance is preserved in the EnTS-tree.

A leaf-node is called *invalid* if its number of entries exceeds a certain value (*overflow*) or if it is empty (*underflow*). A decision-node is called *invalid* if the number of reference-vectors stored in both subtrees together could be stored in one single leaf-node (*underflow*). These three cases are described in section 3.2 in detail and are mentioned here only for completeness (for a short overview see Figure 14).



(a) Solving an overflow of a leaf-node.

(b) Solving an underflow of a leaf-node.

(c) Solving an underflow of a decision-node.

Figure 14: Overview of invalid nodes and the corresponding actions.

A decision-node is called *unbalanced* if the difference in height of its subtrees exceeds a certain value. To solve the unbalance-problem EnTS tries three different strategies (*delegation*, *migration* and *recreation*), which are discussed in the following paragraphs.

Delegation Re-balancing can be seen as the problem of decreasing the height of the deeper subtree. If it is possible to decrease its height, the unbalance is solved as well. Therefore, the systems first attempt is to delegate the problem into the deeper subtree (see Figure 15). If both subtrees have the same height, EnTS tries to decrease the height of both.

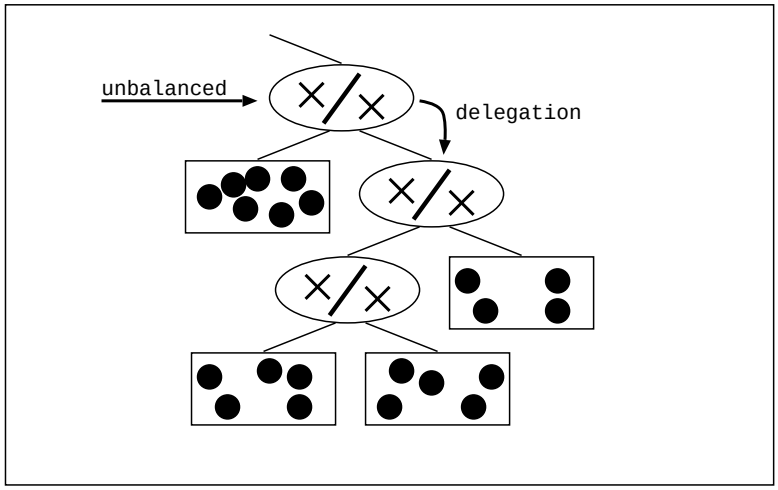


Figure 15: Delegation of the unbalance. The topmost node is unbalanced, but the solving is delegated into its right subtree, because $2^{2-1} * 10 * 0.8 > 14$. It is not delegated further, because $2^{1-1} * 10 * 0.8 \not> 10$. I.e. EnTS tries to decrease the height / (re-)balance the second node.

But first the system checks whether it is worth trying to decrease the height of a subtree or not. There is a chance if:

$$2^{d-1} * C * F > |V|$$

d - depth of the subtree (a leaf's depth = 0)
C - capacity of a leaf-node
F - a factor to prevent floundering
V - set of vectors held in the subtree

The factor F prevents the delegation into a subtree that is only just able to decrease its depth. In this situation it is very likely that the next insertion will lead to an expansion again, i.e. the factor creates a buffer for the next insertion. Delegating the problem as far as possible leads to local (and cheap) repairs.

Migration To decrease the height of a certain subtree (or rebalance it), EnTS tries a migration of all reference-vectors from one side to the other. If all the vectors of one subtree fit into its sibling-tree, the parent-decision is not important any more, because one of its subtrees is empty. This process is shown in Figure 16. It leads to one empty subtree, which can be handled as described in section 3.2 (see Figure 10). I.e. by migrating all the reference-vectors into one subtree and by removing the parent-node, the height of the whole subtree is decreased. To check whether the process has a chance to succeed, the system compares the number of reference-vectors to migrate and the available capacity of the other subtree. If the capacity is bigger, it tries the migration. Unfortunately, it is still not guaranteed that the process does not lead to a split in the other subtree. This could lead to an infinite loop of migrations.

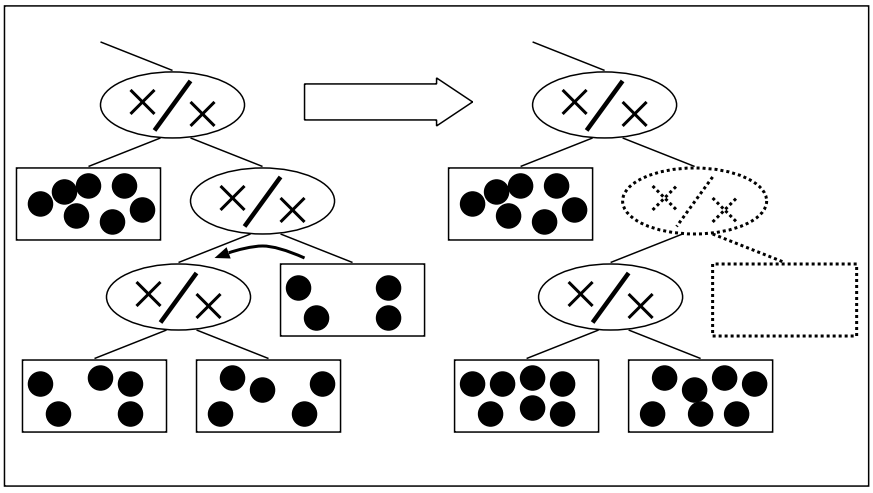


Figure 16: The migration of the reference-vectors from the 2nd level node leads to an empty leaf and therefore a redundant decision-node, which can be removed.

Recreation If neither delegation nor migration are applicable, the system recreates a whole subtree from the scratch. This is done as described in the section 3.1 ("Generation of the Indexing-System").

4 Results

As described in section 1 and 2, the idea was to increase the speed of a certain group of algorithms. But as shown in section 3.3 (No-Man's-Land), this approach might influence the result of the client-algorithms. This section will present some experimental results to show the power of EnTS.

The experiments were done using an EnTS-implementation of the author in Java. All were done multiple times and the average results were used.

4.1 Time-Complexity

To describe the running-time of the EnTS, it is necessary to distinguish two different cases. On the one hand it is possible to describe the speed of the access into the codebook itself. And on the other hand the running time of a client algorithm using an array-implementation and using EnTS can be compared.

Speed of the codebook-access To test the speed of queries into the codebook, codebooks of different sizes were queried several times.

A logarithmic time-complexity is expected for EnTS, due to the internal tree-structure. And a linear one for the array-implementation. The results of the experiment are shown in figure 17.

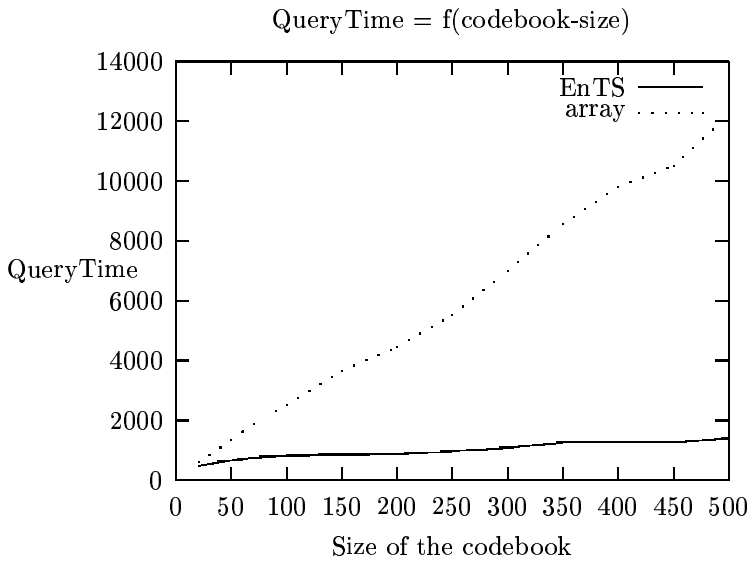


Figure 17: Speed of the access into the codebook.

Speed of client-algorithms To test the influence of EnTS on the speed of the client-algorithm the running-time is compared directly. LBG was used as client-algorithm. Figure 18 shows the results of this experiment.

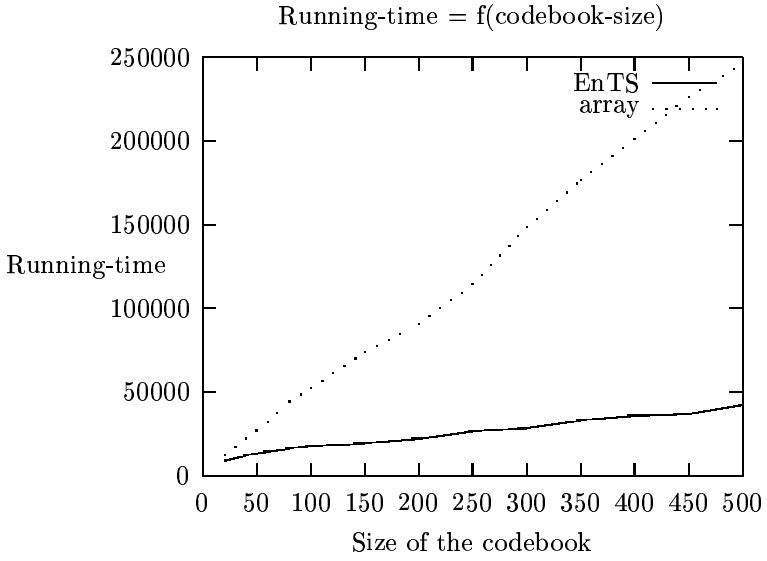


Figure 18: Running-time of a client-algorithm with different codebook-sizes.

4.2 Quality of results and the No-Man's-Land

As described in section 3.3 the quality of the client-algorithm might be influenced by EnTS. The influence is controlled via the No-Man's-Land-parameter described there.

The quality of the client-algorithm is influenced because EnTS might return codebook-vectors which are not the closest for a given input. The theoretically best result can be achieved while working with an array-implementation, because it will always return the very closest codebook-vector.

To measure the quality of the client-algorithm, the distortion of a vector-quantization-algorithm is used. Starting from a fixed size of the codebook and a given number of input-vectors, the experiment was done several times with increasing No-Man's-Land-fractions, ranging from 0.0 to 1.0. The results can be seen in Figure 19.

The experiment showed that the quality is influenced a lot while working with a No-Man's-Land-fraction of 0.0, but differs almost none if this parameter is set to a value greater than 0.4.

4.3 Running-time and the No-Mans-Land

As shown in the last subsection, the quality is influenced, but using a No-Man's-Land-fraction of greater than 0.4, this influence is almost invisible.

Another important influence of the 'No-Man's-Land'-parameter is its impact on the running-time of the client-algorithm. Using a value of 0.0 will increase the speed a lot, whereas a value of 1.0 slows-down the algorithm. To prove this hypothesis the following experiment was done. Using a fixed number of codebook-vectors and a fixed set of input-vectors, an increasing No-Man's-Land-

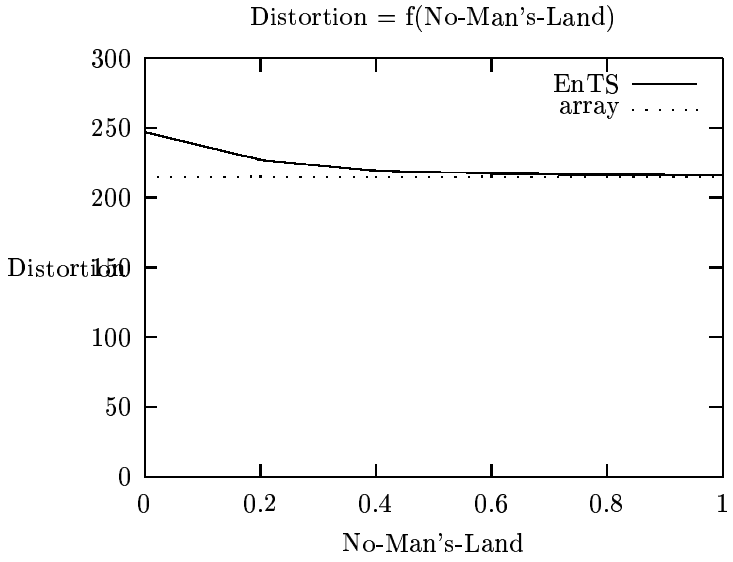


Figure 19: The quality of the client-algorithm dependent on the No-Man's-Land-fraction (smaller distorsion = better quality).

fraction was used ranging from 0.0 to 1.0 As described above (section 4.1), the number of steps performed by the client-algorithm was fixed and the running-times of both implementations were compared.

Figure 20 shows the results of this experiment. The running-time of the array-implementation was constant, but increasing for the EnTS-implementation. The EnT-System is slower if the No-Man's-Land-fraction is greater than a certain value. For this value, the gain in speed is equal to the loss caused by the overhead of the tree-structure.

4.4 Discussion of the other parameters

The experiments described above were done with fixed values for the parameters used for rebalancing. I.e. the parameters which were used to prevent floundering during deletion, merging and rebalancing were kept constant. They were all set to a value of 0.8.

During some experiments it was found that these parameters do not have a significant influence. Only if set to extreme values like 0 or 1, the results were influenced. But to find the best values, further experiments with larger datasets are necessary. The author's recommendation is a value of 0.8.

4.5 Comparison with other algorithms

To compare EnTS with other aproaches in the field of vector-quantisation, some experiments were done which allow the direct comparison with TSVQ, K-tree and k-means as described in [Gev].

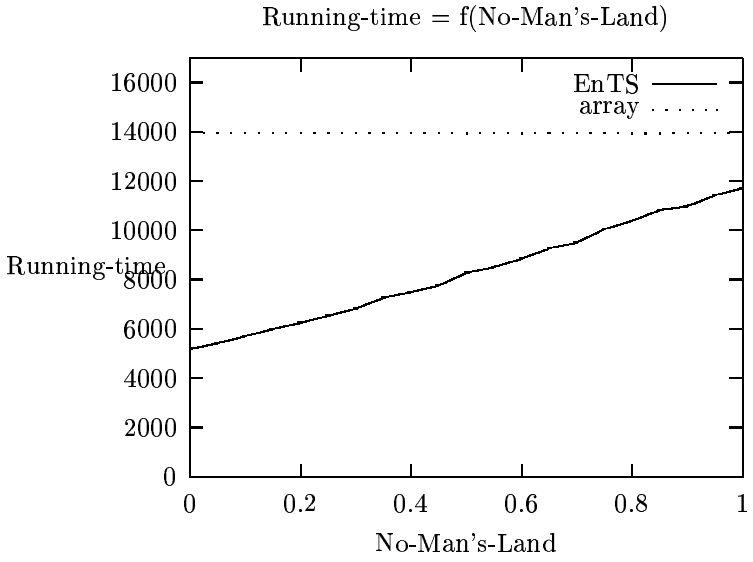


Figure 20: The speed of the client-algorithm dependent on the No-Man's-Land-fraction.

To compare the quality of the algorithms 3924 20-dimensional vectors were quantized and the distortions messured. These vectors come from a speech spectrum and are widely used for similar experiments [oT]. Figure 21 shows the results of the experiment. EnTS outperforms all the algorithms except k-means.

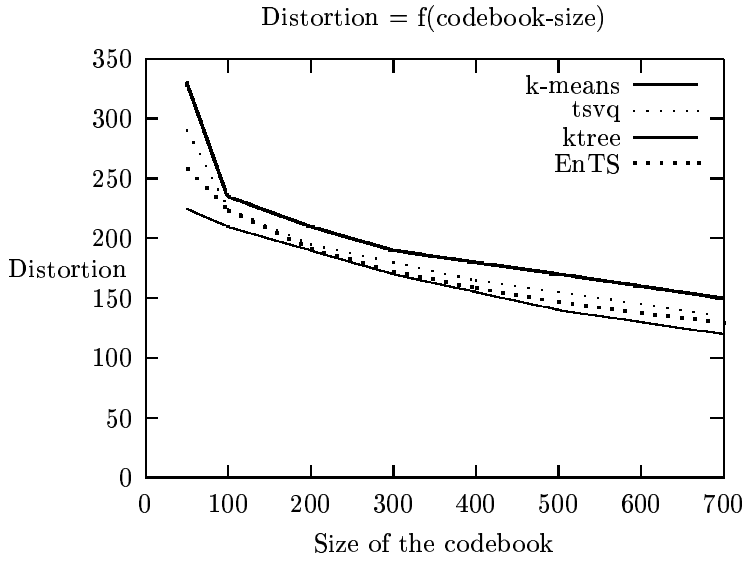


Figure 21: Comparison of codebook distortion

5 Conclusion & future work

As shown in section 4, EnTS provides access into a set of vectors with a time-complexity which is logarithmic in the size of the set. No further requirements are necessary, but the items must belong to a metric vector-space. Furthermore the system is independent of the algorithm it is used in and can be seen as a plug-in for every algorithm working on codebooks.

One drawback of the system is the number of user-parameters. These five parameters might be guessed or determined by experiments, but maybe it is possible to find either good heuristics or replace them with constant values. It should be possible to replace the capacity of the leaf-node-cells and the factors used while modifying the tree (merging, delegation and migration) by constant values. Furthermore, it would be interesting to find the connections between dimension, 'No-Mans-Land', accuracy and speed. A better understanding of these dependencies could lead to a function determining the 'No-Mans-Land'-factor from given constraints. But this requires further experiments with larger data-sets of different dimensions.

The development of more advanced rebalancing-algorithms is another important point for future work. It should be possible to find a way of modifying the hyperplanes to rebalance the tree.

References

[Fri91] Bernd Fritzke. Let it grow—self-organizing feature maps with problem dependent cell structure. In T. Kohonen, K. Mäkisara, O. Simula, and J. Kangas, editors, Artificial Neural Networks, volume I, pages 403–408, Amsterdam, Netherlands, 1991. North-Holland.

[Fri98] B. Fritzke. Vektorbasierte Neuronale Netze. PhD thesis, 1998. `http://pikas.inf.tu-dresden.de/~fritzke/papers/habil.ps.gz`.

[Ger94] K. Rose; D. Miller; A. Gersho. Entropy-constrained tree-structured vector quantizer design by the minimum cross entropy principle. In Martin Cohn James A. Storer, editor, Proceedings Data Compression Conference, pages 12–21. IEEE, IEEE Computer Society Press, 1994.

[Gev] Shlomo Geva. K-tree: A height balanced tree structured vector quantizer.

[Gut84] A. Guttman. R-trees: A dynamic index structure for spatial searching, 1984.

[Koh90] Teuvo Kohonen. The Self-Organizing Map. In New Concepts in Computer Science: Proc. Symp. in Honour of Jean-Claude Simon, pages 181–190, Paris, France, 1990. AFCET.

[MS91] T. Martinetz and K. Schulten. A 'neural gas' network learns topologies, 1991.

[oT] Helsinki University of Technology. Lvq_pak. `http://www.cis.hut.fi/research/som-research/nnrc-programs.shtml`.

[SRF88] T. Sellis, N. Roussopoulos, and C. Faloutsos. Tree: A dynamic index for multidimensional objects, 1988.

[Tol66] J.R.R Tolkien. Lord of the rings, 1966.

[VB96] C. Mohan V. Burzevski. Hierarchical growing cell structures. Technical report, Syracuse University, 1996. `ftp://www.cis.syr.edu/users/mohan/papers/higs-tr.ps`.

[VJH96] J. Austin V. J. Hodge. Hierarchical growing cell structures: Treegcs. Technical report, Dept of Computer Science, University of York, Heslington, York, UK, YO10 5DD, 1996.

"A thing is about to happen which has not happened since the Elder Days: the Ents are going to wake up and find that they are strong." (Gandalf about ents, in [Tol66])