ENTS - A FAST AND ADAPTIVE INDEXING SYSTEM FOR CODEBOOKS

Sebastian Bader

TU-Dresden
Fakultät Informatik
Institut für künstliche Intelligenz
01062 Dresden Germany
s.bader@gmx.net

Frederic Maire

Smart Devices Laboratory
School of SEDC, IT Faculty
Queensland University of Technology
2 George Street, GPO Box 2434
Brisbane Q 4001 Australia
f.maire@qut.edu.au

ABSTRACT

We describe Ents, a new tree structured indexing system for vector quantization. This new indexing system is generic, adaptive and can be used as a software component in any vector quantization system. The cost of this higher speed (compared to tabular indexing) is a negligible degradation of the distortion error. Nevertheless, a parameter allows the user to tradeoff speed for a lower distortion error. A distinctive and attractive feature of Ents is that it can follow a non-stationary input vector distribution by performing local repairs to its indexing tree. Experimental results show that Ents is very fast; it outperforms other tree indexing systems like TSVQ and K-trees.

1. INTRODUCTION

Vector quantization is a powerful computational procedure encountered in a wide range of applications including density estimation, data compression, pattern recognition, clustering, function approximation and time series prediction. One of the major drawbacks of vector quantization is its complexity; the computational cost of finding the nearest neighbours imposes practical limits on the data set size and the rate at which the application can operate.

Tree structures are a common tool to implement indexing systems (for example, R-Trees [5], R^+-Trees [9], K-Trees [4], HiGS [10]). They all provide *find closest* operators with a running-time which is logarithmic in the size of the codebook, by splitting the vector space recursively into smaller nested regions. In other words, the tree structure guides the search at each level into a smaller region containing the input vector. Ents (entropy based tree indexing system) distinguishes itself by its adaptiveness; instead of rebuilding the tree at regular time intervals, local repairs are used to track non-stationary input vector distribution. Moreover a parameter allows the user to tradeoff speed for a lower distortion error.

A fast indexing system should not only provide faster access to the vectors of the codebook compared to a tabular search, but the indexing system should also be independent from the algorithms it serves (k-means, self-organizing maps, neural gas, etc...). From this point of view, Ents is a software component whose interface is composed of a *search* function and three mutator functions. The search function is supposed to retrieve the first closest and second closest codebook vectors to a new input vector. The mutator functions are *insert* (inserts a new codebook vector), *delete* (deletes an existing codebook vector) and *update* (assigns a new location to an existing codebook vector variable). In order to provide a minimal expected running time, the tree of the indexing system should be as balanced as possible. That is, the subtrees of any node should have about the same height.

In section 2, we explain how the indexing tree of Ents is built and kept balanced. Section 3 reports some experimental results on benchmark problems. Section 4 discusses some future work.

The first author worked on this project while visiting the Smart Devices Laboratory, QUT

2. ENTS DESCRIPTION

2.1. Ents Tree Structure

Once abstracted into an interface, an indexing system can be viewed as a black box containing a set of vectors (the codebook) and providing access to them through its functional interface (*search*, *insert*, *delete* and *update*). Classical data structures like B-trees and AVL-trees implement such an interface for totally order sets. Although AVL-trees cannot be used to index data in R^n, as there does not exist a total order in R^n compatible with the arithmetic operators, AVL-trees have inspired the design of Ents. Two types of nodes are used in Ents. The internal nodes of the tree are *decision nodes*; each internal nodes contains a linear discriminant function and two region centres. The leaf nodes are *cluster nodes*; each leaf node contains a small set of codebook vectors (this set is represented with a table, as below a critical number of elements a tree-search is more expensive than a tabular search).

The tree is built by splitting the input-space recursively into half-spaces. The linear discriminant function is chosen so that it distributes evenly the codebook vectors into two half-spaces (maximizes the entropy). To find a good splitting hyperplane the principal component of the codebook (viewed as a set of vectors) is used to determine the weight vector of the linear discriminant function. Once the direction for the separating hyperplane is found, the hyperplane is shifted in such a way as to distribute evenly the codebook vectors on each side of the decision surface. The calculation of the principal component could be done by a PCA. But in Ents a 2-means is performed, and the difference vector of the two means is used as a substitute to the principal component (2-means is computationally cheaper than a PCA). This process generates a shallow and balanced tree, guaranteeing an optimal average access time. Both codebook vector subsets created by this split operation

are processed recursively until each subset becomes small enough to be placed in a single cluster node. The structure of the tree is sketched in Figure 1.

Another way to construct the indexing-system is to build it incrementally. By starting with one empty leaf-node, and inserting the codebook vectors one at a time.

2.2. Ents No-Man's-Land

Sometimes a simple search fails to return the closest codebook vector, as in the end the search for the closest codebook vectors is restricted to the cell of the partition containing the input vector (the partition of the vector space into polyhedral cells is induced by the hyperplanes of the decision nodes). As shown in Figure 1, the closest codebook vector might be contained in a neighbouring cell. To address this *near miss* problem, a *no-man's-land* slice around the separating hyperplane triggers a search in both half-spaces whenever the input vector falls into the no-man's-land. Otherwise, the search in only continued into the half-space that contains the input vector.

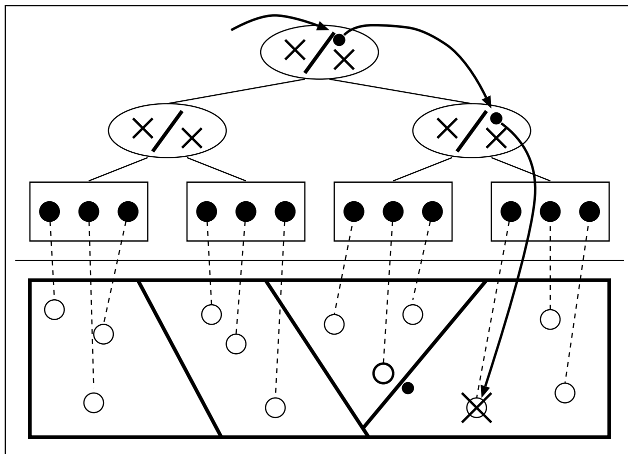

Figure 1: Near miss in a search. The closest codebook vector is the thicker circle, but the returned vector is the crossed circle.

The width of the no-man's-land slice around the separating hyperplane is defined relatively to the distance between the means of the codebook vectors stored in the two subtrees of the decision node. The ratio of the width of the no-man's-land over the distance between the means is a parameter of the indexing system. This user-defined parameter controls the trade-off between the accuracy and the speed of the system. The system is fastest when the no-man's-land has a width of 0, but with this setting the system is more likely to return a codebook vector that is not the closest codebook vector to the input vector. At the other end of the spectrum, an infinitely wide no-man's-land forces the system to consider all codebook vectors. Therefore, the response in this case is much slower, but the system is then guaranteed to always return the correct closest codebook vector. See section 3 for experimental results.

2.3. Balancing in Ents

It is important to keep the tree as shallow and balanced as possible to achieve near optimal average search time. After a sequence of insertion and deletion operations a cluster node (leaf node) might become *invalid* (overflow or underflow). For the same reasons, a decision node might become *unbalanced* (the difference in height of its two subtrees exceeds one). To repair the tree, Ents uses three different strategies; (*delegation*, *migration* and *re-creation*).

In some instances, re-balancing an unbalanced node can be achieved by decreasing the height of the deeper of its two subtrees. If it is possible to decrease the height of the deeper subtree, doing so will solve the balancing problem for the subtree rooted at the unbalanced node. Therefore, Ents first action to rebalance a node is to try to delegate the balancing problem to the deeper subtree (see Figure 2). If both subtrees have the same height, Ents tries to decrease the height of both of them.

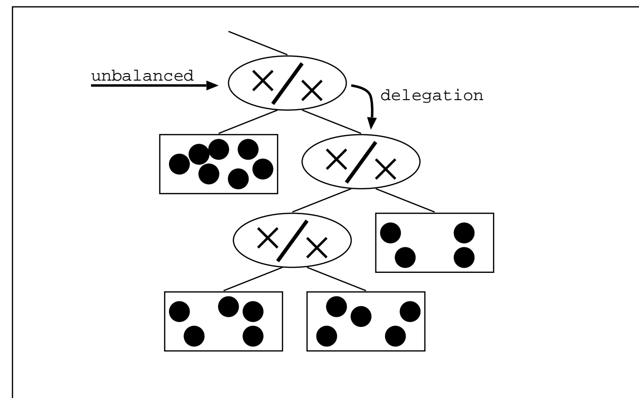

Figure 2: Delegation of the repair. The topmost node is unbalanced. The repair is first delegated to the right subtree, because $2^{2-1} * 10 * 0.8 > 14$. It is not delegated further, because $2^{1-1} * 10 * 0.8 \not> 10$.

The system first checks whether it is worth trying to decrease the height of a subtree or not. It is worth the effort if:

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

d - depth of the subtree
C - capacity of a leaf-node
F - a factor to prevent floundering
V - set of vectors stored in the subtree

The factor F prevents the delegation of the repair to a subtree that is only just able to decrease its depth. If the inequality is not satisfied, it is very likely that a few new insertions will lead to an expansion again. The above test inhibits the reduction of a subtree that would be at almost full capacity if it was reduced. Delegating the repair as far down as possible in the tree leads to local and cheaper maintenance operations.

The second method used by Ents to decrease the height of a subtree (or rebalance it) is to try a migration of codebook vectors from one subtree to another. If all the vectors of one subtree can be added to its sibling-tree, the parent decision node of the subtree loses its raison d'être as one of its subtrees has become empty. This situation is illustrated in Figure 3. By moving all the vectors into one subtree and by removing the parent node, the height of the whole subtree is decreased. To test whether the migration can be achieved, the system compares the number of vectors to move and the capacity of the sibling subtree. If this capacity is large enough, Ents proceeds with the migration.

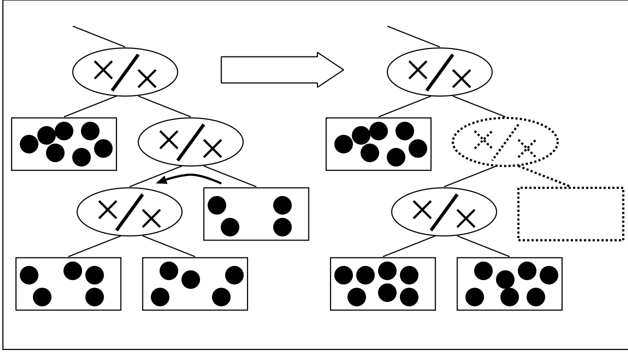

Figure 3: The migration of the vectors from the 2nd level node leads to an empty leaf and a useless decision-node, which can be removed.

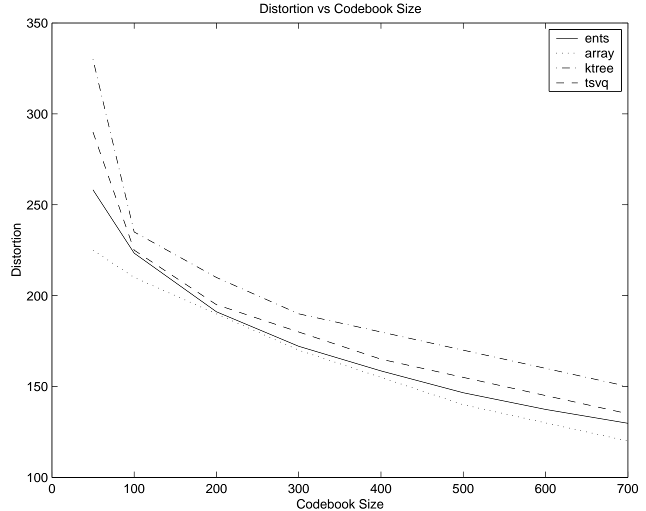

Figure 4: Comparison of distortion errors

If neither delegation nor migration are applicable, Ents rebuilds a whole subtree from scratch.

3. EXPERIMENTAL RESULTS

This section presents some experimental results that demonstrate the usefulness of Ents. The experiments were done using an implementation of Ents in Java. All the experiments were repeated several times (at least 10 times) to obtain statistically significant results.

There are two different ways to assess the performance of an indexing system like Ents. One can evaluate the speed of the codebook by itself (search running time). One can also compare the running time and distortion error of a client algorithm using an array indexing system against a client algorithm using Ents.

3.1. Benchmarking experiments

We compared Ents with TSVQ, K-tree and k-means by running the same experiments as those described in [4].

A data set of 3924 vectors of dimension 20 were quantized using the different quantizers. This data set comes from a speech spectrum benchmark problem [8]. Figure 4 shows the distortion errors of the different systems. Ents outperforms all the algorithms except the array implementation of k-means (which is optimal).

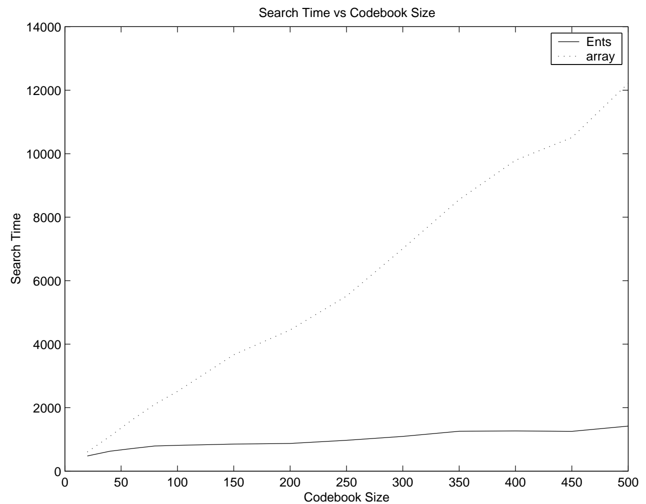

Figure 5: Speed of a nearest neighbour search versus codebook size.

3.2. Time Complexity

We tested how the speed of the search operator scales with the codebook size. As expected, a logarithmic time complexity was observed for Ents (thanks to the tree structure), and a linear complexity was observed for the array indexing system. The results of these experiments are shown in figure 5.

We tested also the incidence of Ents on the speed of the client algorithm running time. LBG was used as the client algorithm. Figure 6 shows the results of these experiments.

3.3. Incidence of the No-Man's-Land Width Ratio on the Distortion Error

As mentioned in section 2, the distortion error of the client algorithm is affected by the indexing algorithm used. Indeed, it might be the case that the indexing system does not return the codebook vectors which are the closest to a given input (the indexing system might return codebook vectors that are relatively close, but not the closest). The best result that can be achieved is obviously obtained with a full tabular search.

In the case of Ents, the negative effect of the tree indexing can be controlled via the No-Man's-Land Width Ratio.

To measure the incidence of No-Man's-Land Width Ratio on the distortion error, we varied the No-Man's-Land Width Ratio from 0.0 to 1.0. The experiment showed that the error distortion initially decreases rapidly with the No-Man's-Land Width Ratio. But the gain becomes much smaller if this parameter is set to a value greater than 0.4. The results can be seen in Figure 7.

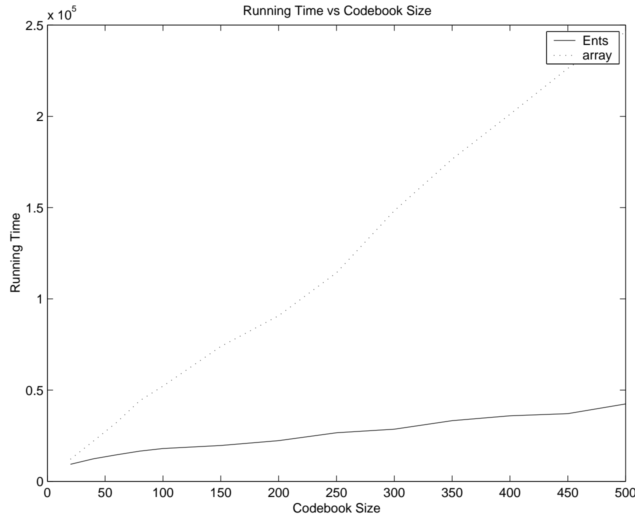

Figure 6: Running time of a client-algorithm versus codebook size.

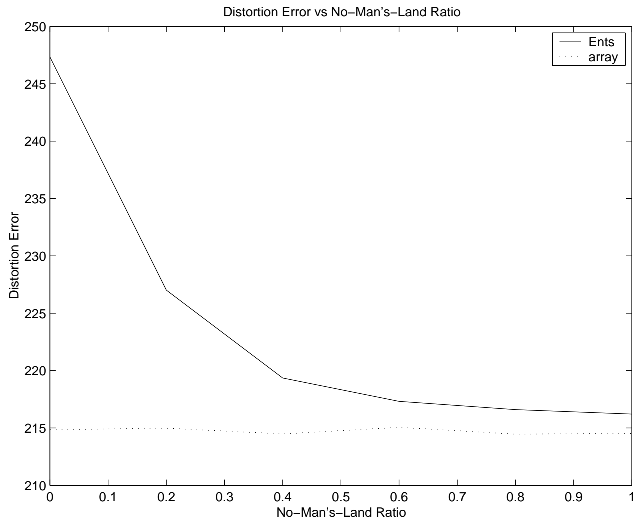

Figure 7: Distortion error versus the No-Man's-Land Width Ratio.

3.4. Incidence of the No-Man's-Land Width Ratio on the Running Time

The cost of increasing the No-Man's-Land Width Ratio is a deterioration in performance with respect to the running time. The user has to find a tradeoff between speed and accuracy. The previous experiment showed that 0.4 was a critical value (almost a turning point) for the No-Man's-Land Width Ratio, as above this value the relative improvement in the distortion will be more expensive with respect to running time.

To quantify the incidence of the No-Man's-Land Width Ratio on the running time, we performed some experiments using a fixed number of codebook vectors and a fixed set of input vectors, and varied the No-Man's-Land Width Ratio from 0.0 to 1.0. We observe that in this range, the running time grows roughly linearly with the No-Man's-Land Width Ratio. Figure 8 shows the results of this experiment.

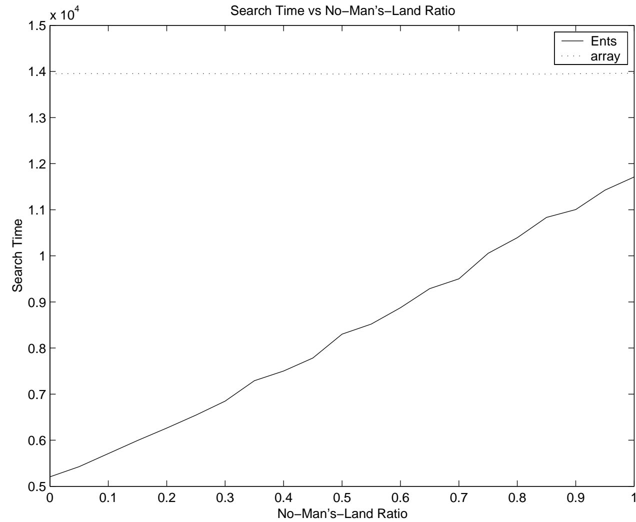

Figure 8: The running time depends linearly on the No-Man's-Land Ratio.

3.5. The Other Parameters

The experiments described above were done with fixed values for the parameters used for the rebalancing algorithms. That is the parameters which are used to prevent floundering during deletion, insertion and rebalancing were kept constant. They were all set arbitrarily to a value of 0.8.

During preliminary experiments it was found that these parameters do not have a significant influence. Only when they are set to extreme values like 0 or 1, do the results differ significantly. To find optimal values, further experiments are necessary.

4. CONCLUSION & FUTURE WORK

Section 3 demonstrated that Ents is a fast indexing system. Moreover, Ents is adaptive and can be made as accurate as desired. From a software engineering point of view, Ents was designed as a component that can be used as a plug-in for any algorithm working with codebooks. The five parameters of the system allow the user to fine-tune the system. However, the default values (that were used in the experiments reported here) should be sufficient.

Ents is a generic indexing system, but its adaptiveness should make it particularly interesting for the machine learning community.

The development of more sophisticated rebalancing algorithms that avoid subtree re-creation would further improve the system.

A Java implementation and a technical report that describes Ents in more details (25 pages) are available at

`http://www.louise15.de/borstel/ents/`

5. REFERENCES

[1] 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.

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

[3] 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.

[4] Geva, S., K-tree: A height balanced tree structured vector quantizer. IEEE Neural Network for Signal Processing Workshop, Sydney, Australia, 2000.

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

[6] Kohonen, T., 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.

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

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

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

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

[11] 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.