# Fast Implementation of the Exact PNN Algorithm

Pasi Fränti
University of Joensuu, Department of Computer Science,
P.O. Box 111, FIN-80101 Joensuu, Finland
Timo Kaukoranta
University of Turku, Department of Computer Science,
Lemminkäisenkatu 14, FIN-20520 Turku, Finland

Turku Centre for Computer Science
TUCS Technical Report No 295
September 1999
ISBN 952-12-0503-2
ISSN 1239-1891

## Abstract

Straightforward implementation of the exact pairwise nearest neighbor (PNN) takes $O(N^3)$ time, where $N$ is the number of training vectors. This is rather slow in practical situations. Fortunately much faster implementation can be obtained with rather simple modifications to the basic algorithm. In the present paper we propose a fast $O(\tau N^2)$ time implementation of the exact PNN, where $\tau$ is shown to be significantly smaller than $N$. We give all necessary data structures and implementation details, and give time complexity of the algorithm both in the best and in the worst case. The proposed implementation achieves the results of the exact PNN with the same $O(N)$ memory requirement.

**Keywords**: vector quantization, codebook generation, clustering algorithms, pairwise nearest neighbor algorithm

**TUCS Research Group**
Algorithmics Group
Image Compression

# 1. Introduction

We consider the codebook generation problem involved in the design of a *vector quantizer*. The aim is to find *M code vectors* (*codebook*) for a given set of *N training vectors (training set*) by minimizing the average pairwise distance between the training vectors and their representative code vectors. There are several known methods for generating a codebook [1]. The most cited and widely used is the *generalized Lloyd algorithm* (GLA) [2]. It starts with an initial solution, which is iteratively improved using two optimality criteria in turn until a local minimum is reached.

A different approach is to build the codebook hierarchically. *Iterative splitting algorithm* [3, 4] starts with a codebook of size 1 where the only code vector is the centroid of the entire training set. The codebook is then iteratively enlarged by a splitting procedure until it reaches the desired size. Another hierarchical algorithm, the *pairwise nearest neighbor* (PNN) [5], uses an opposite, bottom-up approach to the codebook generation. It starts by initializing a codebook where each training vector is considered as its own code vector. Two code vectors are merged in each step of the algorithm and the process is repeated until the desired size of the codebook is reached.

The hierarchical methods have two advantages over the GLA: they can produce codebooks of higher quality [6], and they implicitly generate codebooks of several different sizes during the same run. From the two approaches, the PNN has higher potential because it gives better results with a simpler implementation. The greatest deficiency of the PNN, however, is its slow speed. The method (referred as *the exact PNN*) uses local optimization for finding the code vectors to be combined. A straightforward implementation of this requires an $O(N^3)$ time [7], which is rather slow for large training sets.

Several suboptimal modifications have been proposed in the literature for speeding up the PNN algorithm. Equitz proposed an $O(N \cdot \log N)$ time variant of the PNN, referred as the *fast PNN* [5]. It uses *K-d* tree for localizing the search for the code vectors, and it merges several vector pairs at the same time. The method, however, has not gained as much popularity as the exact PNN, probably because of its more complex implementation and suboptimal results. Another possibility is to generate a preliminary codebook of size $M_0$ ($N>M_0>M$) using the GLA and then apply the exact PNN until the codebook reaches its final size $M$ [8].

In the exact PNN, most of the computation originates from the calculation of the pairwise distances. Since only two code vectors are changed in each step of the PNN, most of the distance calculations are unnecessary. Kurita proposed to store all pairwise distances into a heap structure for reducing the unnecessary calculations [9]. Only $O(N)$ updates are needed after each step of the PNN, each taking $O(\log N)$ time. The method thus performs the exact PNN in $O(N^2 \cdot \log N)$ time but it requires $O(N^2)$ memory, which is impractical for large training sets.

In the present paper we propose a fast $O(\tau N^2)$ time implementation of the exact PNN, where $\tau$ is shown to be significantly smaller than $N$ in practice. The main idea is to maintain a nearest neighbor table, which contains the index of the nearest cluster for each cluster. The optimal cluster pair to be merged can be found by a linear search from the nearest neighbor table. The proposed implementation achieves the result of the exact PNN with rather simple modifications to the basic algorithm. The method requires no complicated data structures and no distance matrix is needed for storing the pairwise distances. The proposed method thus performs the exact PNN using only $O(N)$ memory.

The rest of the paper is organized as follows. The problem formulation and the structure of the PNN are given in Section 2. A fast implementation is introduced in Section 3. Data structures and implementation details are discussed in the same section. Simulation results for various training sets appear in Section 4, and conclusions are drawn in Section 5.

## 2. Pairwise nearest neighbor algorithm

We consider a set of $N$ training vectors ($T_i$) in a $K$-dimensional Euclidean space. The aim is to find a codebook $C$ of $M$ code vectors ($C_i$) by minimizing the average squared distance between the training vectors and their representative code vectors. The distance between two vectors is defined by their Euclidean distance. Let $C$ be a codebook and $P$ the partition of the training set. The distortion of the codebook $C$ is then defined by:

$$distortion(C) = \frac{1}{N}\sum_{i=1}^{N}\left\|T_i, C_{P_i}\right\|^2 \tag{1}$$

where $P_i$ is the partition index of training vector $T_i$. The basic structure of the PNN is shown in Fig. 1. The method starts by initializing each training vector $T_i$ as its own code vector $C_i$. In each step of the algorithm, two nearest clusters ($S_a$ and $S_b$) are then searched and merged. *Cluster* is defined as the set of training vectors that belong to the same partition $a$:

$$S_a = \left\{T_i \middle| P_i = a\right\} \tag{2}$$

The distance (*cost function*) between two clusters is defined as the increase in the distortion of the codebook if the clusters are merged. It is calculated as the distance of the cluster centroids (code vectors) weighted by the number of vectors in the two clusters [5]:

$$d_{a,b} = \frac{n_a n_b}{n_a + n_b}\cdot\left\|C_a - C_b\right\|^2 \tag{3}$$

The cost function is symmetric ($d_{a,b} = d_{b,a}$) and it can be calculated in $O(K)$ time, assuming that $n_a$, $n_b$, $C_a$, and $C_b$ are known. In the following we consider $K$ as constant and thus, the cost function can be calculated in constant time.

The exact variant of PNN applies local optimization strategy: all possible cluster pairs are considered and the one increasing the distortion least (smallest cost function value) is chosen. The clusters are then merged and the process is repeated until the codebook reaches the size $M$. Straightforward implementation of the method takes $O(N^3)$ time because there are $O(N)$ steps in total, and in each step there are $O(N^2)$ cluster pairs to be checked.

```
Set each training vector as a code vector (m=N).
Repeat
    Find two nearest clusters Sa and Sb to be merged.
    Merge the selected clusters; m←m-1.
    Update data structures.
Until m=M.
```

**Fig. 1.** Structure of the exact PNN method.

## 3. Fast implementation of the PNN

The proposed algorithm follows the basic structure of the PNN method but the implementation of the individual steps is slightly different. The main idea is to maintain a *nearest neighbor* table (*NN*), which contains the index of the nearest cluster ($NN_i$) for each cluster $i$. The nearest neighbor for a cluster $a$ is defined as cluster $b$ that minimizes the cost function between $a$ and any other cluster:

$$NN_a = b \text{ if } d_{a,b} \le d_{a,j} \;\forall\; a \ne j, a \ne b \tag{4}$$

Note that the nearest neighbor is not symmetrical, i.e. $NN_a=b$ does not imply $NN_b=a$. In the implementation, it is sufficient to maintain for each cluster the code vector ($C_i$), the cluster size ($n_i$), and the nearest neighbor pointer ($NN_i$) assigned with cost value ($d_i$) indicating the amount of increase in distortion if the clusters are merged. The memory requirement of the data structures is $O(N)$ in total.

### 3.1 Initialization

In the initialization phase, each training vector ($T_i$) is set as its own code vector ($C_i$), and the sizes of the clusters ($n_i$) are set to 1. In order to generate the nearest neighbor table, we need to find nearest neighbor for every cluster. This is done by considering all other clusters as tentative neighbor and selecting the one that minimizes (3). The nearest neighbor pointer ($NN_i$) and the merge distortion ($d_i$) are stored in the nearest neighbor table. There are $O(N^2)$ pairs to be considered in total. Since the cost function can be calculated in $O(1)$ time the time complexity of the initialization phase is $O(N^2)$.

### 3.2 Finding the two nearest clusters

The clusters to be merged are the cluster pair minimizing (3). In our method, this pair can be found by linear search from the nearest neighbor table. The clusters are the one with the minimum $d_i$-value and its nearest neighbor $NN_i$. This operation takes $O(N)$ time.

### 3.3 Merging the clusters

Merging of the two clusters causes changes to all data structures. The merged clusters are denoted in the following by the indices *a* and *b*. The codebook can be updated straightforwardly using the information stored in the data structure. The code vector of the combined cluster is the centroid of the training vectors in the cluster and it can be calculated as the weighted average of $C_a$ and $C_b$:

$$C_{a+b} = \frac{n_a C_a + n_b C_b}{n_a + n_b} \tag{5}$$

Here $n_a$ and $n_b$ are the number of training vectors in the two clusters. The size of the merged cluster is calculated as:

$$n_{a+b} = n_a + n_b \tag{6}$$

The above calculations can be performed in a constant time.

### 3.4 Updating the nearest neighbor table

The key question of the implementation is the maintenance of the nearest neighbor table. It is obvious that the nearest neighbor for the merged cluster must be resolved by considering all other clusters. This can be performed in $O(N)$ time. The rest of the clusters can be classified into two groups:

1. Clusters whose nearest neighbor before merge was *a* or *b*, i.e. $NN_i=a$ or $NN_i=b$.
2. All other clusters.

Only the clusters in the first group need to be updated. The update is performed for a cluster *i* by finding such cluster *j* that $d_{i,j}$ is minimized. This takes $O(N)$ time for a single cluster. An important question is therefore how many clusters belong to the first group. We denote this number by $\tau$.

### 3.5 Amount of necessary updates

We are interested in the question of how many clusters can have the same cluster as their closest neighbor. This is closely related to the *kissing number problem*, which asks the highest number of *K*-dimensional spheres that can be packed so that they all touch one sphere [10]. Unfortunately, the cluster distance function (3) is not Euclidean and therefore the kissing number applies (for certain) only in the initial stage of PNN, and in cases when all cluster sizes are equal. It is thus possible that, in the worst case, the same cluster can be the nearest neighbor for all other clusters, and thus $\tau=O(N)$. This situation could appear when there is one small cluster and all the rest are large. Fortunately this situation is not common in practice and the kissing number (even as an open problem in the general case) indicates that $\tau$ is a function of the vector dimension *K*.

In a favorable case two merged clusters are chosen randomly. Since there is only one nearest neighbor pointer per each cluster, a randomly chosen cluster is the nearest neighbor for one cluster on average. The average number of the updated pointers is therefore $\tau=2$ because there are two clusters involved in the merge operation. This shows that $\tau=O(1)$ in the best case. This approximation, however, is somewhat too optimistic for the average case because the merged clusters are not chosen randomly but they tend to be located in areas of high concentration of clusters.

To sum up, the time complexity of updating the nearest neighbor table is $O(\tau N)$, which is also the time complexity of the entire PNN step. In the best case this is reduced to $O(N)$, but in the worst case it is still $O(N^2)$. The overall time complexity of our method is therefore $O(\tau N^2)$ as there are $O(N)$ steps of the algorithm in total, see Table 1. This compares favorably with the original method since it is most likely that $\tau << N$. The time complexity of the original PNN method is $O(N^3)$ due to the time-consuming nearest neighbor search. Kurita's method requires $O(N^2 \cdot \log N)$ time originating from the update of the heap structure.

**Table 1.** Time complexities of the three variants of the exact PNN: the straightforward implementation of the PNN without any extra data structures (*Original method*), the method that stores all pairwise distances into a heap structure (*Kurita's method*), and the new method using the nearest neighbor table (*Our method*).

| | **Original method:** | **Kurita's method:** | **Our method:** |
|---|---|---|---|
| Initialization phase: | $O(N)$ | $O(N^2)$ | $O(N^2)$ |
| Single merge phase: | | | |
| • Find two nearest | $O(N^2)$ | $O(1)$ | $O(N)$ |
| • Merge the clusters | $O(1)$ | $O(1)$ | $O(1)$ |
| • Recalculate distances | $O(1)$ | $O(N)$ | $O(\tau N)$ |
| • Update data structures | $O(1)$ | $O(N \cdot \log N)$ | $O(N)$ |
| Merge phases in total: | $O(N^3)$ | $O(N^2 \cdot \log N)$ | $O(\tau N^2)$ |
| **Algorithm in total:** | $O(N^3)$ | $O(N^2 \cdot \log N)$ | $O(\tau N^2)$ |

## 4. Test results

We generated training sets from six different images: *Bridge*, *Camera*, *Miss America*, *Table tennis*, *Airplane* and *House*, see Fig. 2. The vectors in the first two sets are 4×4 pixel blocks from the image. The third and fourth sets have been obtained by subtracting two subsequent image frames of the original video image sequences, and then constructing 4×4 spatial pixel blocks from the residuals. Only the first two frames have been used. The fifth and sixth data sets (*Airplane*, *House*) consist of color values of the *RGB* images. Applications of this kind of data sets is found in image and video image coding, and in color image quantization.

For further testing of the parameters $N$ and $K$, we generated subsets of size $N$=(512, 1024, 2048, 4096) from *Bridge* by random sampling of the original set. All sets were further processed by eliminating one dimension of the vectors at a time, in order to generate subsets with various length of vectors ($K$=2..16). In total, 60 different subsets were generated from *Bridge*. The number of code vectors is fixed to $M$=256 in the following tests.

The number of clusters whose nearest neighbor pointer must be updated ($\tau$) may vary from 0 to $N$-2. We studied the number $\tau$ empirically by calculating its average and maximum values when generating codebooks from all 60 subsets of *Bridge*. We observed that the average value of $\tau$ is small in all cases; it varies from 1 to 4 and is practically independent from the size of the training set ($N$). The average value of $\tau$, on the other hand, is an increasing function of $K$ but the growth is still rather slow, see Fig. 3. The average and maximum number of $\tau$ for all training sets are summarized in Table 2.

Following the hypothesis that $\tau$ is independent of $N$, the computation time of our method should be multiplied by 4 when the size of training set is doubled, whereas the running time of the original PNN should be multiplied by 8. This is verified by the observed running times (see also Fig. 4), which are (120, 1082, 8913, 71324) for the original PNN, and (3, 18, 78, 328) for our PNN when $N$=(512, 1024, 2048, 4096). The corresponding multiplication factors for (512→1024→2048→4096) are (9.02, 8.32, 8.00) for the original PNN and (6.00, 4.33, 4.21) for our PNN.

Our method is compared with the Kurita's method in Fig. 5 for subsets of *Bridge* with various number of $K$. The running times of the two methods are rather similar. Usually $\tau$ is smaller than log $N$ but the difference is compensated by the fact that the update of the heap (log $N$) is faster because it does not depend on the dimension $K$. Our method is therefore slightly slower in cases when $K$ is larger, and vice versa. Kurita's method was not applied for larger data sets because of its huge memory consumption.

The observed running times for all training sets are summarized in Table 3. A major speed-up is obtained in all cases in comparison to the original algorithm. The method is about 100 to 350 times faster than the original method for the tested training sets. Since the method evidently gives asymptotic improvement in the average case, the speed-up is greater for large training sets.

The exact PNN is compared in Fig. 6 with two suboptimal variants of PNN. The GLA-PNN method by de Garrido et al. [8] starts with an initial codebook of intermediate size (generated by the GLA), which is then reduced to the final size using the exact PNN. The method is a compromize between higher speed of the GLA and better quality of the PNN. The larger the initial codebook the closer the method is to PNN, and the smaller the initial codebook the closer the method is to GLA. If running time is a critical factor, the GLA-PNN is better choice than Equitz's *K-d* tree variant. Otherwise the exact PNN is recommended.


**Fig. 2.** Sources of the training sets. *The training set *Table tennis* is constructed by random sampling only every fourth block. **The images *Airplane* and *House* are prequantized by 5 bits per each color component.

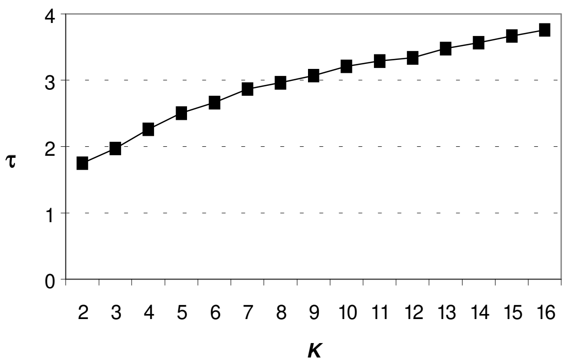


**Fig. 3.** The average number of $\tau$ as a function of $K$.

**Table 2.** The observed numbers of necessary updates ($\tau$).

| Training set: | Average: | Maximum: |
|---|---|---|
| *Bridge* | 3.76 | 16 |
| *Camera* | 3.35 | 16 |
| *Miss America* | 2.44 | 8 |
| *Table tennis* | 4.22 | 23 |
| *Airplane* | 4.61 | 41 |
| *House* | 4.19 | 23 |

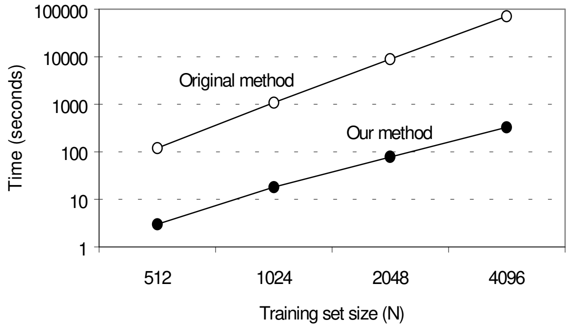


**Fig. 4.** Comparison of running times of the original and our method as a function of $N$ (for *Bridge*).

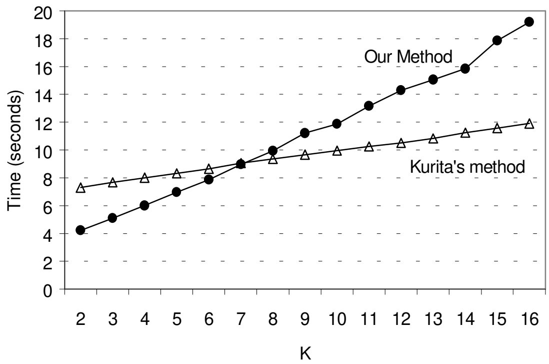


**Fig. 5.** Comparison of running times of Kurita's method and our method as a function of $K$ (for a subset of *Bridge* where $N$=1024).

**Table 3.** Summary of the running times (in seconds).

| Training set | Original PNN | Our PNN | Speed-up factor |
|---|---|---|---|
| *Bridge* | 73,006 | 331 | 221 |
| *Camera* | 73,040 | 300 | 243 |
| *Miss America* | 292,351 | 870 | 336 |
| *Table tennis* | 177,019 | 649 | 273 |
| *Airplane* | 4,751 | 48 | 99 |
| *House* | 2,341 | 27 | 87 |

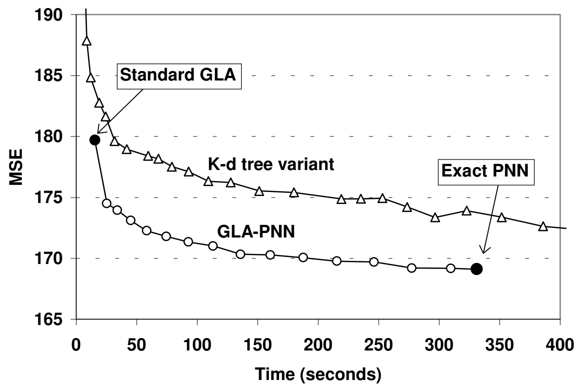


**Fig. 6.** Comparison of the suboptimal PNN variants (for *Bridge*). In the *GLA-PNN* method we apply our $O(\tau N^2)$ implementation of the PNN instead of the original $O(N^3)$ time algorithm. The size of the initial codebook varies from 256 (standard GLA) to 4096 (exact PNN). The *K-d tree* variant is parametrized by varying the maximum bucket size from 8 to 200.

## 5. Conclusions

A fast $O(\tau N^2)$ time implementation of the exact PNN was proposed. The main idea is to maintain a nearest neighbor table for avoiding unnecessary distance calculations. The number of necessary updates ($\tau$) depends on the nature of the data and the dimension of the training vectors ($K$) but it is practically independent from the size of the training set ($N$). In practice, the number $\tau$ was observed to be significantly smaller than $N$ in all training sets, varying from 2.4 to 4.6 for the tested training sets. A speed-up of about 100 to 350 was thus obtained. To sum up, the method achieves the result of the exact PNN using only a fraction of time required by the original algorithm.

The proposed method is also practical and rather simple to implement. It requires no complicated data structures and no distance matrix is needed for storing the pairwise distances. The proposed method performs the exact PNN using only $O(N)$ memory, in contrary to Kurita's method that requires $O(N^2)$ amount of memory. The running times of the two methods were comparable but the proposed method is also applicable for large training sets.

## References


1. A. Gersho and R.M. Gray, *Vector Quantization and Signal Compression*. Kluwer Academic Publishers, Dordrecht 1992.
2. Y. Linde, A. Buzo and R.M. Gray, "An Algorithm for Vector Quantizer Design". *IEEE Transactions on Communications*, **28** (1), 84-95, January 1980.
3. X. Wu and K. Zhang, "A Better Tree-Structured Vector Quantizer". *Proceedings Data Compression Conference*, Snowbird, Utah, 392-401, 1991.
4. P. Fränti, T. Kaukoranta and O. Nevalainen, "On the splitting method for VQ codebook generation". *Optical Engineering*, **36** (11), 3043-3051, November 1997.
5. W.H. Equitz, "A new vector quantization clustering algorithm", *IEEE Transactions on Acoustics, Speech, and Signal Processing* **37** (10), 1568-1575, October 1989.
6. T. Kaukoranta, P. Fränti and O. Nevalainen, "Iterative split-and-merge algorithm for VQ codebook generation", *Optical Engineering*. **37** (10), October 1998. (in press)
7. J. Shanbehzadeh and P.O. Ogunbona, "On the computational complexity of the LBG and PNN algorithms". *IEEE Transactions on Image Processing* **6** (4), 614-616, April 1997.
8. D.P. de Garrido, W.A. Pearlman and W.A. Finamore, "A clustering algorithm for entropy-constrained vector quantizer design with applications in coding image pyramids". *IEEE Transactions on Circuits and Systems for Video Technology* **5** (2), 83-95, April 1995.
9. T. Kurita, "An efficient agglomerative clustering algorithm using a heap". *Pattern Recognition*, **24** (3) 205-209, March 1991.
10. J.H. Conway and N.J.A. Sloane, *Sphere Packing, Lattices and Groups*. Springer-Verlag, New York, 1988.

Turku Centre for Computer Science
Lemminkäisenkatu 14
FIN-20520 Turku
Finland

http://www.tucs.abo.fi/

University of Turku
• Department of Mathematical Sciences

Åbo Akademi University
• Department of Computer Science
• Institute for Advanced Management Systems Research

Turku School of Economics and Business Administration
• Institute of Information Systems Science